2

HEADQUARTERS AIR FORCE INSPECTION AND SAFETY CENTER

SYSTEM SAFETY HANDBOOK

AFISC SSH 1-1

SOFTWARE SYSTEM SAFETY

5 SEPTEMBER 1985

TABLE OF CONTENTS

No of Printed Pages: 41
OPR: SESD
Approved by: Col Paul D. Smith
Editor: Ellyn Eisenbeisz
DISTRIBUTION: X

Department of the Air Force
Headquarters Air Force Inspection and Safety Center
Norton Air Force Base, California 92409-7001

AFISC SSH 1-1
5 September 1985

SOFTWARE SYSTEM SAFETY HANDBOOK

1. PURPOSE

The primary purpose of this handbook is to document technical knowledge of safety techniques and methodologies than can be used to support acquisition programs which involve computer/embedded computer systems. It is intended to aid in the development of "safe" system software. This handbook does not and will not describe how to design functional performance into a system. Rather, the handbook does and will continue to describe design choice limits, boundary values, and preferred practices that relate to maximizing overall system safety. The major emphasis of this handbook is to provide an assist in specifying and designing for system safety. The section herein that provides a checklist of rules and guidelines is aimed at the up-front and top-down design principles. A later section describing verification and evaluation techniques is aimed at picking up where specification and design implementation perfection leave off. Some verification and evaluation techniques can serve early in the design process, even before hardware and software is built. Others serve better after software is built (with or without target hardware). This handbook supplements the MIL-STD-882B software hazard analysis task.

1.1 CONTRACT APPLICATION

This handbook, of and in itself, is not construed or implied to be a contractual document. The procuring activity must extract the applicable handbook rules and guidelines and include them in the system specification. Specific tasks and data items must be called out in the Statement of Work (SOW) and identified in the Data Item Descriptions (DID's).

2. SCOPE

This document is intended for use primarily by DOD program managers and technical specialists in the areas of safety and software engineering. It is intended to serve as a companion document to MIL-STD-882 and to act as a guide in accomplishing the software safety task.

2.1 ORGANIZATION OF THIS HANDBOOK

Following this introduction, this document has six major sections and three appendices. Section 3 contains the specific DEFINITIONS which will be used throughout the document. Section 4, INTRODUCTION, begins with a rationale for software safety programs and the philosophical background and outlook necessary for a successful program. Section 5, DESIGN GUIDELINES FOR SOFTWARE SYSTEM SAFETY, delves into the specific requirements necessary to design safety into software systems and to insure that the software design is analyzable. Section 6, SOFTWARE SAFETY ANALYSIS, addresses the general philosophy of the three major stages of software safety analysis. Section 7, SOFTWARE SAFETY ANALYSIS TECHNIQUES, discusses the various techniques used for this analysis. Section 8, SOFTWARE SYSTEM SAFETY DESIGN RULES AND GUIDELINES, contains design requirements and guidelines which can be used to develop safe software. Appendices A through C contain examples of a software analysis specification, a request for proposal (RFP) and a statement of work. Please note that the appendices are examples only and are not designed to be used as written, nor do they imply any preferred analysis methodology! They all must be carefully tailored to reflect the analysis method(s) desired and the system on which it (they) are to be applied!

3. DEFINITIONS

For definitions not specifically listed below, the following sources, in order of preference, is as follows:

a. DOD-STD-1679A Military Standard Software Development

b. IEEE Standard Glossary of Software Engineering Terminology

3.1 Built-in-test (BIT). A hardware and/or software-implemented check intended to detect prespecified acceptable conditions (or deviations therefrom) and to take some prespecified course of action (i.e., warning, alert, shutdown) upon detection of an anomaly.

3.2 Code Walk-through. A manual simulation of predefined test cases (i.e., test input and expected output) led by the programmer and addressed to a small (e.g., 3-6) group of qualified personnel, preferrably personnel not directly involved with the program being reviewed. (See also Inspection)

3.3 Design Walk-through. A step-by-step detailed examination of the design of the software by a small group of qualified personnel.

3.4 Detailed Level Analysis. Additional to flow of control, i.e., making sure each data item is correctly referenced, computed or used.

3.5 Development, Software. The engineering process and effort that results in software, encompassing the span of time from initiation of the software requirements/design effort through delivery to and acceptance by the contracting agency.

3.6 Erroneous Bit. A single bit in a register or memory location that was intended to be a "1" which was interpreted as a "0" (or vice versa) during program execution.

3.7 Error, Software. An occurrence, or lack thereof, during the execution of a program that prevents satisfaction of the specified software requirements, fails to perform as designed, or performs a function not required and/or not desired.

3.8 Fault Tree. An analytical technique, whereby an undesired system state is specified and the system is then analyzed in the context of its environment and operation to find all credible ways in which the undesired event could occur.

3.9 Firmware. Software which resides in non-volatile computer memory that is not alterable by the computer system during program execution. For the purpose of system safety, firmware is to treated as any other piece of software.

3.10 Flow Chart/Diagram. A graphic representation of the processing order or execution of instructions and subroutines in which symbols are used to represent operations, data, flow, equipment, etc.

3.11 Graphic Representations. A tool used to facilitate the translation of requirements into software design or software design into source code. They are intended to be comparable to blue prints for hardware. Examples are program design languages [PDL/Ada (Ada is the trademark of the U.S. Government, Ada Joint Program Office)], (Structured English, Pseudo-Code, Pidgin-English, etc.), functional flow diagrams, block diagrams, etc.

3.12 Hardware. Physical parts of a system, such as mechanical and electrical components, switches, and input/output devices.

3.13 Inspection. A step-by-step examination of the program, lead by the programmer and addressed to a small group of qualified personnel, preferably personnel not directly involved with the program being reviewed. During the examination each step is checked against a predetermined list of criteria.

3.14 Interface Analysis. An analysis of module interfaces and associated variables.

3.15 Machine Code. Instructions that are intended to be input directly into the instruction register of a computer's central processing unit (CPU).

3.16 Microprocessor. The central electronic device which actually executes the software. (Usually surrounded by peripheral devices such as memory, buffers, decoders, etc., which allow interaction with the rest of the system involved.)

3.17 Mnemonic Operation Codes. Computer instructions written in a meaningful human notation, i.e., ADD, STO, etc., but which must be converted on a one-to-one basis into machine language for computer operation.

3.18 Node. A point where several paths meet.

3.19 Object Code. Compiler or assembler output that is itself executable machine code or that can be processed to produce executable machine code. For the purpose of software analysis, object code is normally analyzed by looking at its mnemonic representation. This mnemonic code can be produced by compilers which produce intermediate assembler code or through a disassembly process.

3.20 RAM. Random access memory.

3.21 Requirements Walk-through. A step-by-step, detailed examination of the requirements of the software.

3.22 ROM. Read only memory.

3.23 Safety Critical. Those software operations that, if not performed, performed out-of-sequence, or performed incorrectly could result in improper control functions (or lack of control functions required for proper system operation) which could directly or indirectly cause or allow a hazardous condition to exist.

3.24 Soft Tree. A term coined to describe a fault tree which is constructed on a system which includes a software interfacing with hardware. A software fault tree.

3.25 Software. A combination of associated computer instructions, statements, and computer program data definitions required to enable the computer hardware to perform computational or control functions. Note: This definition includes firmware within its applicability. This definition of software is independent of the type of physical storage media in which the software resides.

3.26 Software Element. A constituent part of the system's software. The following would be examples of software or program elements: programs, routines, modules, tables, or variables.

3.27 Software Reliability. The probability that the software will execute for a particular period of time without a failure, weighted by the cost to the user of each failure encountered.

3.28 Software Safety. The application of system safety engineering techniques to software in order to insure and verify that the software design takes positive measures to enhance system safety and that errors which could reduce system safety have been eliminated or controlled to an acceptable level of risk.

3.29 Software Safety Analysis. A hazard evaluation technique which identifies software commands (with and without errors) either singularly or in combination with hardware responses that could result in safety critical hazards.

3.30 Source Code. A computer program written in a language designed for ease of expression of a class of problems or procedures by human. A generator, assembler, translator, or compiler is used to then translate the source program into object code for use by the computer.

3.31 System. The equipment, facilities, software, operator, maintainer, and technical data associated with a single entity (weapon system).

3.32 System Allocation Documents. Various specification and requirement documents (requirements/performance specifications) which represent the flow of system-level requirements into subsystems.

3.33 System Level Analysis. Analysis concerned with flow of control, and data items and variables influencing it.

4. INTRODUCTION

The development and usage of microprocessors and computers has grown at a phenomenal rate in recent years, from 1955, when only 10 per cent of our weapon systems required computer software, to today, when the figure is to over 80 per cent. This increased reliance on computers has put the aerospace industry and the military in the position of controlling hazardous systems with computers and associated software. Examples of this include the Minuteman and Peacekeeper weapon systems and the F-16 fly-by-wire system. This use of computers, while improving system performance, has also complicated the job of the safety engineer as methods to ensure the safety of the computer-controlled aspects of the systems has lagged behind the development of the systems.

Traditionally, in safety analyses, the computer subsystem and the software it contains have been considered a "black box" that gives a specific output based on a given input and has not been considered in the safety analysis process. However, as the use of software-controlled systems has grown, so has the incidence of mishaps directly attributable to software increased. It has become very apparent that software can no longer be ignored from a safety standpoint.

4.1 THE SOFTWARE HAZARD

It is well understood that software in and of itself is not intrinsically hazardous; however, once the software is associated with a system, it becomes as potentially hazardous as the total system. For a hazard to occur, there must be some undesired/uncontrolled release of energy, energy normally contained by some hardware component(s). The failure, malfunction, or error in control of that hardware must cause or allow the hazard to occur; therefore, normally only software which exercise direct command and control over the condition or state of the hardware components or can monitor the state of the hardware components are considered critical from a safety viewpoint. Software which controls and monitors systems, such as missiles or aircraft, are considered to be hazardous. However, other systems that provide indirect control or data for critical processes, such as the attack warning system at NORAD or flight navigation systems, must also be considered as hazardous, as erroneous data can lead to potentially hazardous erroneous decisions by the human operators or companion systems.

Software hazards fall into four broad categories: (a) inadvertent/unauthorized event, an unexpected/unwanted event occurs; (b) out-of-sequence event, a known and planned event occurs but not when desired; (c) failure of event to occur, a planned event does not occur (e.g., a hazard is allowed to propagate because the program does not detect the occurrence of the hazard or fails to act); (d) magnitude or direction of event is wrong, this is normally indicative of an algorithm error.

Traditionally, the failure of the computer has been viewed only from a standpoint of electronic component failures in reliability and safety analyses. The electronics can be analyzed and mean time between failures (MTBF) assigned. However, an MTBF cannot be directly applied to software as software does not fail. The "software failures" are actually errors built into the program. These errors fall into two main classes: (a) incorrect or incomplete specifications and requirements which leads to incorrect or incomplete designs, or (b) software errors made during programming or coding. A third failure class is hardware related; software errors occur due to a hardware-induced corruption of the program. Software changes due to the effects of radiation bombardment of RAM fall into this category.

The fact that software does not fail does not make the issue of safety analysis easier. In theory, a properly designed testing program could detect all errors. In practice, however, this becomes virtually impossible due to the many possible combinations of sequences and the timing of those sequences with respect to one another. Further, modern software testing includes time compression techniques which tend to make error windows so small that the window may not be hit and the errors, therefore, undetected. Additionally, as tests tend to reflect the requirements to which the software was written, errors in the requirements are seldom detected by testing.

In the past short history of software development, much of the testing or debugging has been accomplished by trial and error. When an error was detected, the software code was analyzed and a change was made. This process took time, and latent software errors often lay unnoticed during development only to appear after the product had been fielded. Modern weapon systems cannot afford to be programmed by such methods, as latent errors can lead to system destruction.

4.2 MANAGEMENT CONSIDERATIONS

Software safety, which is a subset of system safety, is a relatively new field and is going to require a conscientious effort by all those involved in any system acquisition process or development effort to insure it is adequately addressed during system development. Hardware and software engineering efforts must be accomplished as a team effort. Engineers must attend their counterpart's specification and design reviews so that they are fully aware of hardware/software and other interfaces and how each of their efforts are interdependent and how they jointly affect safety. System safety personnel must have (or have access to) the appropriate hardware and software expertise and must also always maintain a "total systems" viewpoint.

A special effort must be made to insure that appropriate requirements are transmitted to the contractors. A good up-front effort to define a system safety theme and insure that proper safety requirements are contained in requests for proposals (RFPs) and specifications will save time, money, and possibly lives over the system lifetime. Just as a good system safety program needs to begin when a project is started, so must the efforts to insure software safety. Specific up-front requirements imposed upon the software development effort will help insure that safety is designed into the software and that the required safety analyses can be easily and inexpensively accomplished.

5. DESIGN GUIDELINES FOR SOFTWARE SYSTEM SAFETY

5.1 SYSTEM CONSIDERATIONS

The foremost factor in insuring system safety is to insure that the entire system be looked at as a system, not separate hardware and software elements. At the very inception of a system development effort, a system safety theme needs to be identified which will specify how fault tolerant the system must be and how the system is to handle faults; i.e., must the system work at least at some level of operation or, given a certain class of fault must the system shut itself down or initiate some specific procedure? This theme will then serve as a general guide as to hardware and software design requirements and is one of the first steps in specifying system safety requirements.

The next step is to insure that appropriate requirements are included in the RFPs, statements of work, contracts, and the system and subsystem specifications. In addition to the system specific requirements, the following documents of the issue in effect as of the date of the contract should be included in contract requirements:

TABLE 5.1. APPLICABLE DOCUMENTATION

DOCUMENT NUMBER	DOCUMENT NAME
MIL-STD-882	System Safety Program Requirements
MIL-STD-483	Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs
MIL-STD-490	Specification Practices
DOD-STD-1679	Software Development Standard
DOD-STD-480	Configuration Control - Engineering Changes, Deviations and Waivers
AFR 122-10 (if applicable)	Nuclear Safety Design and Evaluation Criteria

5.2 HARDWARE CONSIDERATIONS FOR SOFTWARE SAFETY

If software has anything which even resembles a failure mode, it is in the area of hardware-induced failures. Just as when a hardware component fails and causes other component or system failures, the same effect also applies to software. A hardware failure can cause a bit error in a computer word which induces a software failure because the software instruction no longer has its original meaning. The failure mechanism is entirely within the hardware, but the software is erroneous because it has a new and unintended meaning. These failures may be permanent or a temporary glitch due to effects such as alpha particle radiation. Hardware design and software should attempt to take these problems into account and provide a method of flagging them and reducing their effect (e.g., parity check on memory data and software commands prior to acceptance). It should be noted that a BIT will probably not catch a soft error (glitch). Also, hardware sensor failures can provide erroneous inputs. When operated on by the software, erroneous judgements are made, resulting in improper commands being issued. Again, the failure mechanism is entirely in the hardware, but the software outputs are erroneous and defeat proper system operation. Therefore, credible hardware failure modes must drive design requirements, both of the hardware and of the software.

5.3 REQUIREMENTS FOR THE VARIOUS PHASES OF SYSTEM DEVELOPMENT

5.3.1 Specification Phase

The development of system and subsystem specifications is one of the most crucial activities during a system development effort. This is especially true for system safety and specifically for software safety. Without appropriate and adequate safety requirements in the development specifications, software may neither provide the desired degree of safety nor even be analyzable. In general, the software must be designed to an acceptable level of risk, given that failures, errors, and adverse environmental conditions will occur. This requires that a general software safety philosophy be decided early in the program and be so stated in the requirements documents. Also, to help insure that developed programs are analyzable, DOD-STD-1679, DOD Software Development Standard, or similar structured programming requirements should be levied upon the contract.

The safety effort must begin by defining both general and specific safety requirements for the top-level system specifications. These requirements must then be flowed down to the lower-tiered specifications. As this flowdown process takes place, there must be increased attention placed on the specific safety functional requirements of the individual subsystems. For instance, hazards identified in the preliminary hazard analysis may drive specific requirements for inclusion in a computer program configuration item (or even how a particular problem must be solved in rare cases).

Safety critical functions and elements must be identified early and appropriate safety design criteria provided for them in the specifications. These criteria should include possible methods for risk avoidance (e.g., safety kernel techniques or the use of safety assertions) and possibly an ordered precedence of various methodologies (e.g., program design inhibits, interlocks, hardware configuration, or operational plans and procedures). Additionally, requirements for entrancy/reentrancy constraints, output monitoring, and hardware status checking must be specified and the requirement for testing/analysis called out.

The two greatest difficulties encountered at this stage of development are that many of the actual requirements are not known and that system/subsystem interfaces are not accurately described or understood. Compounding these difficulties are system designs which push the state-of-the-art on one or more technological fronts. To help overcome these difficulties, one of the best tools available is "history!" Similar programs or activities should be actively reviewed to glean lessons learned data. From a generic basis, the following have already proven to be safety concerns:

a. Routines which disable interrupts should be considered safety critical.

b. Conditional statements must be reviewed to insure that conditions are correctly stated and that the conditions can and will be appropriately met (e.g., conditions that call for an exact match of floating point numbers can easily result in an infinite loop).

c. What conditions can result in a divide by zero or what are the effects of other analogous singularities in the program? Not only can a divisor of zero cause this problem but also "very small" numbers may be truncated to a zero by the software.

d. Priority of fault detection and safing/correction logic vs normal processing "in line" logic.

e. Can the system and routines handle the amount and frequency (load) of data? Additionally, is the precision and scaling of data appropriate and what conditions could cause a hazardous condition?

f. With systems utilizing technologically advanced ideas, IR&D and proof-of-design laboratory demonstration hardware figures heavily in the proposed developmental design. Consequently, basic building block safety provisions must be pressed into such laboratory/exploratory designs, even though "system" safety at that point has little meaning.

5.3.2 Design Phase

Once the specifications have been written, the next task is to translate those requirements into an actual design. It is important to again stress that safety is the joint responsibility of all those concerned on a development effort. Engineering must translate specification safety requirements into the actual design, and with the quality and safety offices, insure the accuracy of algorithms, interfaces, and other requirements that could affect system safety. The selected safety philosophy, as well as structured programing techniques, should be strictly adhered to. Software critical items (see PHA) shall follow modular design techniques and shall be separated from and not be intertwined within non-critical items.

The software hazard analysis effort (described in section 6) should begin early during software development by analyzing the various program design documents, such as flowcharts, program structure documents, etc., for compliance with safety requirements and to identify potentially hazardous conditions. It is important to stress that the software cannot be looked at in a vacuum. It is just one part of the overall system, and a mishap can occur when it directly or indirectly influences a hazardous system action or activity. Closely related to the safety analysis effort is the need for a strong configuration control program which will identify changes to critical software elements which have been previously examined.

5.3.3 Coding Phase

As during the previous design phase, system safety is the responsibility of all concerned with the design process. The coding effort must be reviewed by software-knowledgeable quality assurance personnel through code walk-throughs, and other methods to insure accuracy, compliance with software engineering and safety design requirements, and to identify errors. Safety personnel will continue the software safety analysis by examining the source/target machine executable code of safety critical elements and insuring the independence of these modules from noncritical elements. These personnel will also ascertain that the program performs its intended functions correctly, and that it does not perform any unintended functions (e.g., no extraneous code). Independent testing of segments of code must be completed prior to the segment being incorporated into the main code package.

5.3.4 Testing Phase

From a safety viewpoint, complete and thorough testing of a system may neither practical nor desirable due to the costs and/or resources involved. However, testing must show that all hazards that have been identified have been either eliminated or controlled to an acceptable level of risk. Safety testing must verify that not only will the system function normally within specified environments but also test for reaction to failures and boundary conditions and abnormal/out-of-bound environments (e.g., 'negative' 'what-if' testing). Safety inhibits, traps, interlocks, assertions, etc., must be verified by test or simulation. It is vital that all code within the program be executed at least once and each decision brought to each of its possible outcomes at least once, although no necessarily in every combination.

Depending upon the type and importance of the system, independent safety and verification and validation should be considered.

5.3.5 Deployment/Operational/Maintenance Phase

Software safety does not terminate with the deployment of a system. All complex software programs have "bugs", many of which are not found until after the system becomes operational. The way in which those bugs are fixed determines whether or not the software is or remains "safe". The single most dangerous act in software maintenance is to allow a machine language patch to a single module without carefully analyzing the impact of that change to the rest of the software. Sometimes a seemingly innocuous change can cause system wide devastation. All changes, not just those to safety critical code, should be reviewed for impact to the safety of the overall system.

Changes to baselined software documentation and codes are normally controlled, processed, and implemented in accordance with a software configuration management system. Software safety needs to ensure that special emphasis is given to the safety aspects of changes that are handled in such a manner. In order to carry out this effort, all changes to previously analyzed specifications, requirements, designs, codes, and test plans, procedures, cases, or criteria are subjected to a software hazard analysis.

The beginning point of this change hazard analysis will depend upon the highest level within the documentation that is affected by the changed being proposed. For example, when the change affects the Software Top Level Design Document, the analysis of the change begins with the top-level design analysis software safety tasks defined earlier. Appropriate subtasks under that analysis and all subsequent analysis subtasks should be performed until either:

a. All appropriate subtasks have been completed, or

b. It is determined that the change neither creates a new hazard nor impacts on one which has already been resolved.

There are also software safety considerations regarding the installation of firmware changes during these phases. When the software for a system is contained in firmware (ROMs), and changes for the software are fielded as new ROMs, consideration must be given to ensuring that the ROMs are inserted properly, and in the correct order. Even when the firmware is distributed as full occupied circuit cards, consideration must be given as to whether multiple cards can be swapped, or whether a card can be inserted backwards. If the software is distributed as magnetic media (tapes, disks, floppies, etc.), extra consideration must be given to the proper installation.

Besides just proper physical installation, there also must be a method of verifying that the changes are being made on a system at the intended modification level. If other modifications are/have been issued, how does the field unit ensure that the software changes being inserted are made to a system that has the right modifications, or that has not been already modified? Software changes may not be readily apparent above the code level. This is especially important if the change is issued as a partial set of ROMs, and does not involve a complete swap out. If several modifications are/have been made to the software in this fashion, sooner or later one modification will be made before the previously issued one (or one will make it through the paperwork before the other). Since all software depends on all of the other software in a system, this can have disastrous effects.

Some good rules for software field maintenance include the following:

a. Whenever possible, issue firmware changes as fully populated and tested circuit cards.

b. If fully populated cards cannot be issued, then issue complete ROM sets, numbered and/or color coded. Analyze the software for the effects of improper insertion (one or more swaps), and provide a method for field verification of proper insertion (BIT or checksum).

c. If a complete set of ROMs cannot be issued, then ensure that no safety critical code modules cross the address space between old and new ROMs. That is, the safety critical code should be entirely contained in the new firmware or in the old firmware. There should be no cases in which part of a safety critical module is in the old firmware and part is in the new firmware. Field verification of the proper installation and integrity of the complete system software is essential. Make sure that the difference between subsequent modifications can be detected by BIT or checksum.

d. If the changes are issued as magnetic media, include an automatic patching utility which checks to ensure that it is patching the proper revision level of the software, and which then changes the revision level of the installed software.

e. If, as a last resort, or if the changes are very minor, the software is issued as written instructions, they should be inserted using a checksum verification utility.

6. SOFTWARE SAFETY ANALYSIS

6.1 PRELIMINARY AND FOLLOW-ON SOFTWARE HAZARD ANALYSES

Software hazard analysis is accomplished so that the total system will operate at an acceptable level of risk. To be effective, analysis must be started early enough to actively influence the design of both the software and the total system. The analysis effort is started when the system allocation process has delineated the responsibilities between the hardware and software. The software hazard analysis must be structured to permit continual revision and updating as system and software design matures.

6.1.1 Preliminary Software Hazard Analysis

The preliminary software hazard analysis is a direct offshoot of the system preliminary hazard analysis (PHA). As it is normally cost prohibitive to analyze all system software to the same depth, the system PHA, when integrated with the requirements levied upon the software, will identify those programs, routines, or modules that are critical to system safety and must be examined indepth. These software elements shall be identified as safety critical. However, all software must be analyzed at least to the extent necessary to determine any impact or influence upon the safety critical code. All programs, routines, modules, tables, or variables which control or directly/indirectly influence the safety critical code shall also be classified as safety critical. Additionally, some or all of the software tools (e.g., compilers, support software, etc.) may also have to be designated as "critical". All safety critical software elements will be analyzed to the source/object code level by the follow-on software hazard analysis.

The preliminary software hazard analysis is accomplished by analyzing the following documents:

a. System and subsystem preliminary hazard analyses

b. System and subsystem specifications.

c. System allocation and interface documents.

d. Functional flow diagrams and related data.

e. Flow charts or their functional equivalent.

f. Storage allocation and program structure documents.

g. Background information related to safety requirements associated with the contemplated testing, manufacturing, storage, repair, and use.

h. System energy, toxic and hazardous event sources which are controlled or influenced by software.

6.1.2 Follow-on Software Hazard Analysis

The follow-on software hazard analysis is a continuation of the preliminary software hazard analysis and begins when coding of the software begins, normally following the critical design review (CDR). Those software elements that have been previously identified as being safety critical shall be analyzed at the source/machine executable code level. The level of effort required depends on the perceived risks. In certain instances, if the source code is written in a high order language and there is a high level of system risk involved, the run time object code should be analyzed to insure that the compilation or interpretation process has not introduced any hazards or negated any safety design efforts.

Additional activities that occur during the follow-on analysis include a review of all hardware/software, software/software, and software/operator interfaces and of critical data (e.g. files, etc.). Analysis is accomplished on all algorithms, and calculations for correctness and input/output/timing sensitivity. While some of these activities may fall under the purview of QA and/or reliability, those elements affecting safety critical elements must be reviewed by system safety. Also, system safety must continue its active monitoring of the design and coding effort, with special attention being placed on design/program changes. When a program is submitted for analysis, it should also be placed under in-house configuration control so that the analysis report will reflect a known program version, and if changes are made, safety will know of the changes.

7. SOFTWARE HAZARD ANALYSIS METHODOLOGIES

Although not an exhaustive list, the following techniques can be used to help provide a thorough software hazard analysis. Additionally, a thorough software hazard analysis of a system may require application of more than one of the following techniques or others not yet identified in this document.

7.1 SOFTWARE FAULT TREE (SOFT TREE)

Before the topic of the software fault tree can be discussed, a brief description of fault trees in general is needed. The following three paragraphs are taken from the US Nuclear Regulatory Commission NUREG-0492, "Fault Tree Handbook," January 1981:

"The fault tree analysis can be simply described as an analytical technique, whereby an undesired state of the system is specified (usually a state that is critical from a safety standpoint), and the system is then analyzed in the context of its environment and operation to find all credible ways in which the undesired event can occur. The fault tree itself is a graphic model of the various parallel and sequential combinations of faults (or system states) that will result in the occurrence of the predefined undesired event. The faults can be events that are associated with component hardware failures, human errors, or any other pertinent events which can lead to the undesired event. A fault tree thus depicts the logical interrelationships of basic events that lead to the undesired event--which is the top event of the fault tree.

It is important to understand that a fault tree is not a model of all possible system failures or all possible causes for system failure. A fault tree is tailored to its top event which corresponds to some particular system failure mode, and the fault tree thus includes only those faults that contribute to this top event. Moreover, these faults are not exhaustive--they cover only the most credible faults as assessed by the analyst.

It is also important to point out that a fault tree is not in itself a quantitative model. It is a qualitative model that can be evaluated quantitatively and often is. This qualitative aspect, of course, is true of virtually all varieties of system models. The fact that a fault tree is a particularly convenient model to quantify does not change the qualitative nature of the model itself."

7.1.1 The Soft Tree

The term soft tree has been coined to describe a fault tree which includes software interfacing with hardware. The software fault tree proceeds in a manner similar to hardware fault tree analysis and uses a subset of the symbols currently in use in the hardware counterparts. Thus, hardware and software trees can be linked together at their interfaces to allow the entire system to be analyzed. This is extremely important since software safety procedures cannot be developed in a vacuum but must be considered as part of the overall system safety. For example, a particular software error may cause a mishap only if there is a simultaneous human and/or hardware failure. Alternatively, the environmental failure may cause the software error to manifest itself. The soft tree also can help identify credible fault modes within the computer system, such as flag or register failures, which will cause the software to act in an undesired manner.

The first step in the software fault tree is conducting a PHA analysis to identify and categorize the hazards posed by the system. Classifications range from "catastrophic" to "negligible." Once the hazards have been determined, fault tree analysis proceeds. It should be noted that in a complex system, it is likely that not all hazards can be predetermined. This fact does not decrease the necessity of identifying as many hazards as possible but does imply that additional procedures may be necessary to insure system safety.

The goal of software fault tree analysis is to show that the logic contained in the software design will not produce system safety failures. A further goal is to determine environmental conditions which could lead to these software induced failures. The basic procedure is to assume that the software will lead to a catastrophe and then to work backward to determine the set of possible causes for the condition to occur.

The root of the fault tree is the event to be analyzed, i.e., the "loss event." Necessary preconditions are described at the next level of the tree with either an AND or an OR relationship. Each subnode is expanded in a similar fashion until all leaves describe events of calculable probability or are unable to be analyzed for some reason. Further information on the general construction of fault trees can be found in NUREG-0492 and specific information on soft trees can be found in "SOFT TREE, Fault Tree Techniques as Applied to Software" by the Directorate of Systems Safety, Armament Division, Air Force Systems Command, Eglin Air Force Base.

Soft tree/fault tree analysis can be used at various levels and stages of software/ system development and the level of node development can also be tailored depending upon how the tree is to be used and the corresponding stage of software/system design. Thus, the analysis can proceed and be viewed at various levels of abstraction from trees derived from program flows or a program design language (PDL) to final trees whose nodes are at the code level. It is, therefore, desireable to build the initial trees early during the software life cycle so that safety can be designed into the software during its development. (Software fault trees can also utilize the network tree data base generated during a software sneak analysis.)

When working at the code level, the starting place for the analysis is the code responsible for the output. The analysis then proceeds backward deducing both how the program got to this part of the code and determining the current values of the variables. It is at this level that processor and computer faults can also be identified. For example, a soft tree event could read "carry flag set." The input to this event, as a minimum, should include an OR gate with at least two inputs: (1) carry occurred due to some calculation, and (2) carry flag bit failed high. Again, the final goal of the analysis is total system safety, and this will require that the analysis be done by persons knowledgeable of the computer equipment and languages being used. As with any fault tree, the safety engineer must know the system he has to analyze and make every attempt to model the system accurately.

As a final note, to facilitate this analysis, as well as system safety, software should be developed such that the majority of the safety critical functions, decisions, and algorithms are within a single (or few) software development modules. This, as well as a structured approach that protects against inadvertent jumps, etc., will significantly reduce the scope of work that is needed on a given tree as the tree will only need to extend down to the code level for only extremely critical parts of the software.

7.1.2 Uses of Fault Trees

7.1.2.1 Cutset Analysis

Boolean (i.e., logic or circuit) algebra may be applied to the fault tree to find the minimum cutsets, which are all combinations of basic events (component-level and often quantifiable events) which will cause the top event. The algebra removes the intermediate logical interrelationship levels of the tree, converting the tree to an equivalent form which is in terms of basic events only.

The cutsets show which single-point failures, double-point failures, and higher-order failures are possible and which lead to occurrence of the top event. The number of events in each cutset shows the amount of inhibits, in different combinations, on the occurrence of the top event.

Cutset analysis computer programs are available, such as FTAP, SETS, PREP, ALLCUTS, and ELRAFT.

7.1.3.2 Quantitative Analysis

If component failure rates and hazard duration times are available for each basic event, a probability of the top event may be calculated.

Conversely, given a desired probability of the top event, allowable time for operation of the system may be calculated.

In addition, basic events or cutsets may be ranked quantitatively as to their relative importance to the top event.

This quantitative analysis of the fault trees is traditionally a hardware analysis technique where a "component" is easily defined. Experimental attempts are being made to define software components and to determine failure rates for these.

Computer programs are available for quantitative analysis, including IMPORTANCE, BACSIM, KITT, SAMPLE, MOCARS, and FRANTIC.

7.1.2.3 Common Cause Analysis

Common cause analysis deals with dependencies between events in a cutset. For example, a cutset may contain three events, "transistor X shorts," "transistor Y shorts," and "transistor Z shorts." If all three events occur, the top event will occur (the system will fail). If all three events can be triggered by an external common cause, such as heat, and if heat is present, the system will fail.

In the case of software, common cause analysis can be applied to cutsets to determine environmental effects on top events. For example, if the contents of two registers are thought to be affected by a certain environment, and both are in the save cutset, the cutset is a more important possible system failure mode when that environment is present.

Computer programs, such as COMCAN, BACKFIRE, and SETS, are available to aid in the sorting of events in cutsets to locate common susceptibilities.

7.2 SOFTWARE SNEAK CIRCUIT ANALYSIS

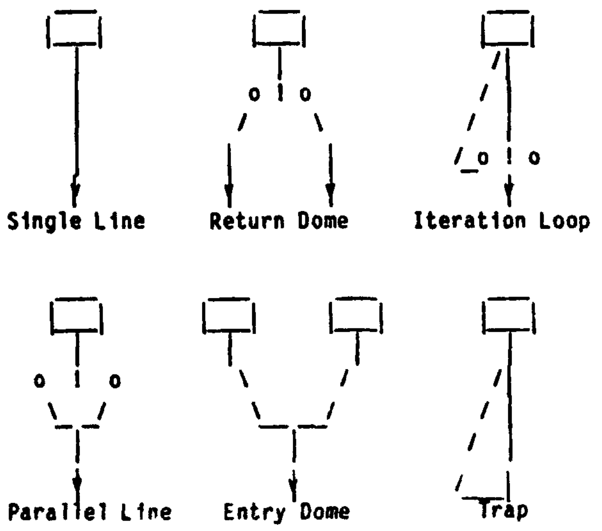
Five specific software analysis techniques are described in the material that follows. The techniques are basically static analyses. Some of the information on analysis techniques other than software sneak analysis has been obtained from "Checkout Techniques, Software Reliability Guidebook," Prentice-Hall, 1979; Robert L. Glass.

Software sneak analysis: Data used for software sneak analysis should reflect the program as it is actually written. This includes system requirements, system description, coding specifications, detailed and complete source code, a compilation listing, and operating system documentation. All records are written against these documents.

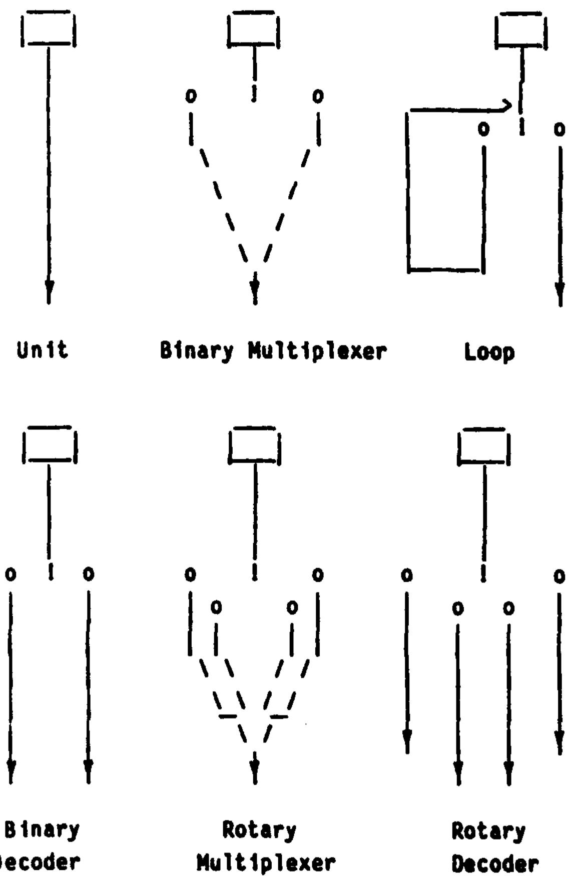
Software Sneak Analysis is used to discover program logic which causes undesired program outputs or inhibits a desired output. The technique involves the reduction of the program source code to topological network tree representations of the program logic.

Direct analysis of program listings is difficult because the system is modular for ease of programing. Also, the code is listed as a file or record, without regard to functional flow. The first task of the software sneak analyst is to convert the program source code into a form usable for analysis. In most cases, this step requires computer conversion. In either case, the program source code is converted with reference to an input language description file into topological network trees such that program control is considered to flow down the page.

Once the trees have been drawn, the analyst identifies the basic topological patterns that appear in the trees. Six basic patterns exist: the single line, the return dome, the iteration/loop circuit, the parallel line, the entry dome, and the trap circuit, as shown below:



In a system level analysis, code is modelled in terms of impedances, powers, grounds, switches, nodes, and relay coils and switches. Trees are constructed hierarchally, so that the analysis proceeds from the top down. In a system level analysis, the following topographs are identified:



Although at first glance a given software tree may appear to be more complex than these basic patterns, closer inspection will reveal that the code is actually composed of these basic structures in combination. As each node in the tree is examined, the analyst must identify which pattern or patterns include that mode. The analyst then applies the topograph specific clues that have been found to typify the sneaks involved with that particular structure. These clues are in the form of questions that the analyst must answer about the use and interrelationships of the instructions that are elements of the structure. These questions are designed to aid in the identification of the sneak conditions in the instruction set which could produce undesired program outputs.

Software sneaks are classified into four basic types:

a. Sneak output - the occurrence of an undesired output.

b. Sneak inhibit - the undesired inhibition of an output.

c. Sneak timing - the occurrence of an undesired output by virtue of its timing or mismatched input timing.

d. Sneak message - the program message does not adequately reflect the condition.

When potential sneak is identified, the analyst must verify that it is valid. The code is checked against the latest listing. Compiler information may be reviewed concerning the language in question. If the sneak is verified, a software sneak report (SSR) is written which includes an explanation, system-level impact, and a recommendation for elimination of the sneak.

During the course of analysis, questionable programing practice or instruction implementations may be encountered and are reported in a software design concern report (SDCR).

If two or more documents or the program listing and a supporting document do not agree, the program listing is assumed to be correct. If the analyst then determines that the condition is not a software sneak or questionable practice, a software documentation error report (SDER) is issued for the discrepancy.

The following paragraphs describe various methods of analysis used in the software analysis:

a. Desk checking. Desk checking is one of the earliest forms of software verification. It involves:

(1) Reviewing a program listing for faults and compliance to requirements,

(2) Performing arithmetic calculations to verify output value correctness, and

(3) Manually simulating program execution in order to understand and verify program logic and data flow.

Since desk checking is such an ill-defined concept, it is difficult to provide a cost estimate for its use. It is undoubtedly true, however, that moderate amounts of desk checking save more money than they generate in cost.

Desk checking efforts concentrate on areas of special problems, especially suspected errors or code inefficiencies, and involve techniques appropriate to that problem.

b. Code Walk-through. Code walk-through is a process by which a team of programing personnel (i.e., technologists) do an indepth review of a program, or portion of a program, by inspection. In general, the participants are verbally lead sequentially through the logic flow of the program as represented in the listing. (Note: the leader should not be the responsible programmer) All logic branches should be taken at least once. The function of each statement will be discussed as it is encountered. Program requirements and design specifications will be present for correlation of function to its driving factors.

The walk-through should not occur until after coding of the program to be reviewed is completed, well annotated, and syntactically correct.

c. Structural analysis. A structural analyzer is an automated tool that seeks and records errors in the structural makeup of a computer program undergoing analysis. Structural analysis is a relatively new concept, beginning in the early 1970s. Structural analyzers are almost always language and installation or project specific. Most structural analyzers built to date accommodate only FORTRAN or COBOL. For example, DAVE, built at the University of Colorado, processes CDC-6000 FORTRAN programs looking for uninitialized variables via a very elaborate algorithm.

The major factor in the cost of a structural analysis is the acquisition of a structural analyzer. Costs can range from trivial, if the program is already in the public domain, to upwards of $100,000 for implementation of an elaborate analytical tool.

d. Proof of correctness. Proof of correctness is the process of using mathematical theorem-proving concepts on a computer program or its design to show that it is consistent with its specification. This is done by breaking the program into logical segments, defining input and output assertions for each segment, and demonstrating that, when the program functions, if all input assertions are true, then so too are all output assertions. It must also be shown that the program successfully terminates.

Many researchers are currently working in the proof-of-correctness area. Small algorithms and programs have been proven in this environment; however, it has yet to be demonstrated on programs of any significant size.

Even for small simple programs, the symbolic manipulations involved in the proof-of-correctness technique can be overly complex, introducing errors into the computation of the statements to be proven as well as the proof of those statements. Thus, the technique would be most successful on highly mathematical and relatively straightforward program segments.

Lack of practical experience with proof of correctness makes it difficult to quantify costs. Usage costs are significant, possibly adding 100 to 500 percent to the cost of the portion of the software being proven. However, by lowering the projected mishap number, the life cycle cost of the system could be substantially reduced.

Specification requirement for sneak analysis: The sneak analysis specifications are currently listed in MIL-STD-785B, Reliability Program for Systems and Equipment Development and Production.

7.3 NUCLEAR SAFETY CROSS-CHECK ANALYSIS (NSCCA)

NSCCA is a rigorous methodology developed exclusively to satisfy the requirements of Air Force Regulation 122-9 and is accomplished by an agency or contractor who is independent from the program developer. (Some names of activities and plans vary depending upon the branch of service but the intent remains the same) The methodology, to a great degree, is an adversarial approach to software analysis in that its basic objective is to show, with a high degree of confidence, that the software will not contribute to an undesirable event.

The NSCCA process has two components: technical and procedural. The technical component evaluates the software by analysis and test to assure that it satisfies the system's nuclear safety objectives. The procedural component implements security and control measures to protect against sabotage, collusion, compromise, or alteration of critical software components, tools, and NSCCA results. (Note: While this method was originally designed to meet the needs of nuclear systems, the method can be tailored and applied where ever desired.)

7.3.1 Technical Component of NSCCA

The technical component begins with a criticality analysis which maps the nuclear safety objectives against the discrete functions of the software system. Qualitative judgments are made to assess the degree to which each function affects the nuclear safety objectives, and suggestions for the best methods to measure the software functions are made. The program manager uses the criticality assessment to decide where best to allocate resources to meet the requirements, then the NSCCA program plan is drawn. The program plan establishes the tools and facilities requirements, analyses requirements, test requirements, test planning, and test procedures. This family of documents establishes in advance the evaluation criteria, purpose and objectives, and expected results for specific NSCCA analyses and tests. Establishing these details beforehand promotes the independence of NSCCA and avoids "rubber-stamping."

7.3.1.1 Derivation of Nuclear Safety Objectives (NSO)

The first step in criticality analysis is derivation of the NSO. NSOs are derived from DOD Directive 5030/15 and AFR 122-10. DOD 5030/15 directs that, for nuclear weapons systems, positive measures be taken to:

a. Prevent nuclear weapons involved in accidents or incidents from producing a nuclear yield.

b. Prevent deliberate prearming, arming, launching, firing, or releasing of nuclear weapons, except upon execution of emergency war orders or when directed by competent authority.

c. Prevent inadvertent prearming, arming, launching, firing, or releasing of nuclear weapons.

d. Insure adequate security of nuclear weapons.

These four DOD safety standards define the undesirable events, detonation, prearming, arming, launching, firing, releasing, and loss which are to be protected against. The fourth standard is also satisfied in the procedural component of NSCCA.

AFR 122-10, Nuclear Safety Design and Evaluation Criteria, breaks down the four DOD standards into specific requirements which are the minimum positive measures necessary to demonstrate that nuclear weapon system software is predictably safe. The combination of DOD safety standards and the design criteria form the menu of NSO.

7.3.1.2 Functional Breakdown of the System Software

Step two of the criticality analysis is to break down the total software system into the lowest order of discrete functions or modules. It is intuitively obvious that the more modular a system is the more easily analyzable it becomes. Once the system is broken down to the lowest order of function, each function is analyzed to determine the degree to which it operates on or controls a nuclear critical event (i.e., prearming, arming, etc.). Modules which have no influence on the nuclear critical events are not included for further analysis. A matrix is then prepared plotting software modules against the NSO and an influence rating given (high - h, medium - m, low - l). Also prospective evaluation techniques are shown. With the criticality analysis in hand, program management can determine the best course of action to pursue the NSCCA. Figure 7.3.1 shows an example of a criticality analysis.

Software					Evaluation Method		
Modules/ Data	1	2.	i.	n	Exam	Test	Demo
F1	h	l	m		X	X	
F2	m	l	m				X
.							
.							
Fi	l	h	m	l	X		X
.							
.							
Fn	m	l			X	X	X

Figure 7.3.1

7.3.1.3 Conduct of the NSCCA

7.3.1.3.1 NSCCA Program Plan

The program plan identifies detailed NSCCA activities and schedules compatible with the software development contractor's schedule of reviews and delivery dates. The plan also contains man-loading estimates for each activity.

7.3.1.3.2 NSCCA Tools/Facilities Plan

The tools/facilities plan identifies software tools to be used in each analysis task. All required tools are described. For new or modified tools, it provides schedules for development and identifies quality assurance methods to be applied to the tool development process. The plan also describes requirements for test facilities and schedules for testing.

7.3.1.3.3 NSCCA Test Requirements

The test requirements document, which is derived from the system and software requirements specifications, identifies all performance and safety requirements to be analyzed and tested. It also identifies requirements imposed on the test tools in order to properly evaluate system end-item requirements. Test requirements are initially derived from specification documents by the criticality analysis. In addition, hidden safety requirements implied by the coupling of requirements or system constraints are identified and recorded for analysis and test.

7.3.1.3.4 NSCCA Test Plan

The test plan identifies the scope of testing to be applied to each nuclear critical performance requirement identified in test requirements. The scope of each test is defined by establishing an analysis or test scenario for each requirement.

7.3.1.3.5 NSCCA Test Procedures

The test procedures identify how the scope of testing described in the test plan will be achieved. For each test scenario, detailed procedures are developed specifying the input, tool(s), and expected output from the test. The test procedures begin after completion of the test plan and after review of the detailed program design and coding. As errors or deficiencies are detected in the program source code, specifications, interface documents, data base description documents, or other design related documents, they are recorded on discrepancy reports.

7.3.1.3.6 NSCCA Final Report

The final report summarizes the NSCCA activities and results and presents conclusions and recommendations regarding program performance and safety. The final report concludes whether the system sufficiently complies with its performance and safety requirements to warrant certification for operational use.

7.3.2. Procedural Component of NSCCA

Nuclear weapons system software subjected to NSCCA ascends to the Air Force Critical Components List and, as such, comes under the provisions of AFR 122-4, Nuclear Safety: The Two-Man Concept. The objective of this program is to protect the software from any environment which could subject it to possible compromise or alteration. To establish equal confidence in the NSCCA, the same level of protection must apply to the analysis environment. A high level of confidence is achieved by instituting strong personnel, document, and facility security measures, coupled with stringent product control procedures.

7.3.2.1 NSCCA Personnel Security

Special precautions are taken to insure the trustworthiness of participants and the integrity of results. Special background investigations are conducted for all project personnel who participate in NSCCA activities. Each participant in formal NSCCA analyses or test will be cleared for SECRET and CNWDI before commencing work. In addition, project engineers will be cleared for TOP SECRET, COMSEC, and cryptographic information.

7.3.2.2 NSCCA Document Security

Each of the completed documents used or produced during NSCCA is kept under configuration control to insure only current data are distributed. Distribution lists are maintained for each document so that all initial recipients receive any updates or changes. Master copies of documents are retained in the project files with classified documents stored in approved GSA containers.

7.3.2.3 NSCCA Test Facility Security

Rigorous security controls are exercised during interim and formal NSCCA testing to maintain the integrity of machine-readable program code and test data and to prevent unauthorized access to classified information. All classified code and data are stored on clearly marked physical media (disks and tapes) which is secured in GSA-approved containers when not in use. During classified operations, access to the facility is restricted to appropriately cleared personnel with an established need to know. Terminals external to the facility are physically detached from the computers to prevent unauthorized access to classified information. During formal NSCCA operations, two-person control restrictions apply to NSCCA testing performed in the facility.

7.3.2.4 NSCCA Product Control

Strict configuration management is exercised for software items used during interim and formal NSCCA. This includes successive versions of the system software, qualified test tools, prescribed input data sets, and program patches. Each qualified test tool is kept under configuration control so that analysis and test results will be valid and repeatable. During formal NSCCA activities, two-person control is implemented over all nuclear critical program materials to prevent their inadvertent or unauthorized alteration. Critical documentation items, such as performance specifications, design specifications, interface control documents, test plans, test procedures, as well as controlled software (includes program source and load tapes, program patches, and qualified software tools), are stored in two-person controlled containers when not in use. An audit scheme is established to preserve two-person control as materials are used. Whenever two-person controlled items are removed from storage, they are kept under surveillance by two authorized persons at all times. All tape or disk manipulations are performed in such a manner that both people can insure that the tape or disk is not altered. Comparisons and spot checks are performed periodically to insure that working copies and controlled copies are identical.

7.4 SAFETY ANALYSIS USING PETRI NETS

7.4.1 A Petri net (invented by Carl Petri in 1961) is a mathematical model of a system. One of the advantages of Petri nets over other models is that the user describes the system using a graphical notation and thus need not be concerned with the mathematical underpinnings of Petri nets. Some other potenital advantages of using Petri net models include:

They can be used early in the development cycle so that changes can be made while sill relatively inexpensive.

A system approach is possible with Petri nets since hardware, software, and human behavior can be modeled using the same language.

Petri nets can be used at various levels of abstraction.

Petri nets provide a modeling language which can be used for both formal analysis and simulation.

Adding time and probabilities to basic Petri nets allows incorporation of timing and probabilistic information into the analysis.

The model may be used to analyze the system for other features besides safety.

Petri nets have been used primarily to analyze performance and "correctness" (especially with regard to synchronization problems in concurrent systems), however, it has been proposed that they can be used to analyze system designs for safety and fault tolerance. Unlike the fault tree, the safety analysis can be accomplished by a computer without human guidance because the design is first represented as a mathematical system. Furthermore, if a complete analysis becomes impractically large, the model can be executed on the computer in a simulation mode and important information about the design derived. For example, faults can be injected and the result examined. Finally, timing analysis may be more easily accomplished with a Petri net model than with fault trees.

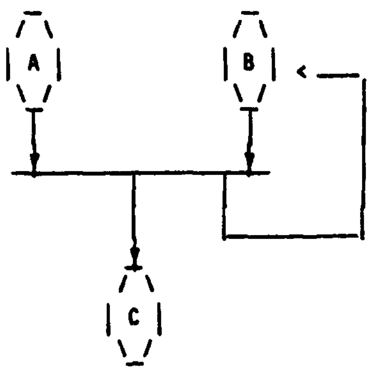


Figure 7.4.1

In a Petri net, the system is modelled in terms of conditions and events. If certain conditions hold, then an event or "state transition" will take place. After the transition, other (or the same) conditions will hold. A condition (called a "place" in Petri net terminology) is represented by a circle "O" and an event or transition by a bar. An arrow from a condition (or place) to a transition indicates that the condition is an input (or pre-condition) to the transition. Similarly, a post-condition is indicated by an arrow from the transition to the condition. Figure 7.4.1 shows a simple example. It says that if conditions A and B hold, then the transition can "fire." After it fires, then conditions B and C will hold.

The Petri net model can be "executed" to see how the design will actually work. "Tokens" represented by a dot "•" are used to indicate that a condition is currently true. Thus if a condition holds, it will contain a token. In the example in Figure 7.4.2, transition t_1 cannot fire because P_1 does not have a token. But all the pre-conditions for t_2 are currently true (i.e. P_2 and P_3), so t_2 can fire. Figure 7.4.3 shows the next state of the Petri net. When t_2 fires, the tokens are removed from the pre-conditions and a token is deposited in each of the post-conditions. In the example, the only post-condition is P_5. Note that it is possible for the same condition to be a pre-condition and a post-condition for a transition. If a transition has no pre-conditions, then it can fire at will.

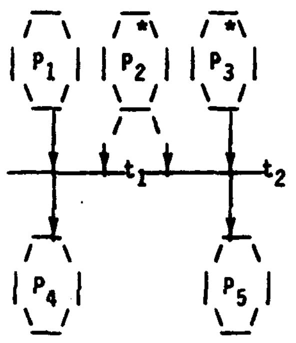


Figure 7.4.2

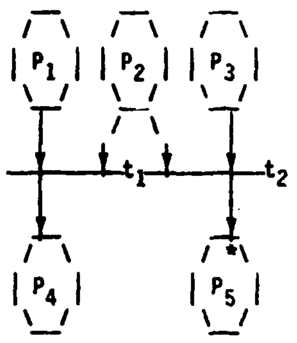
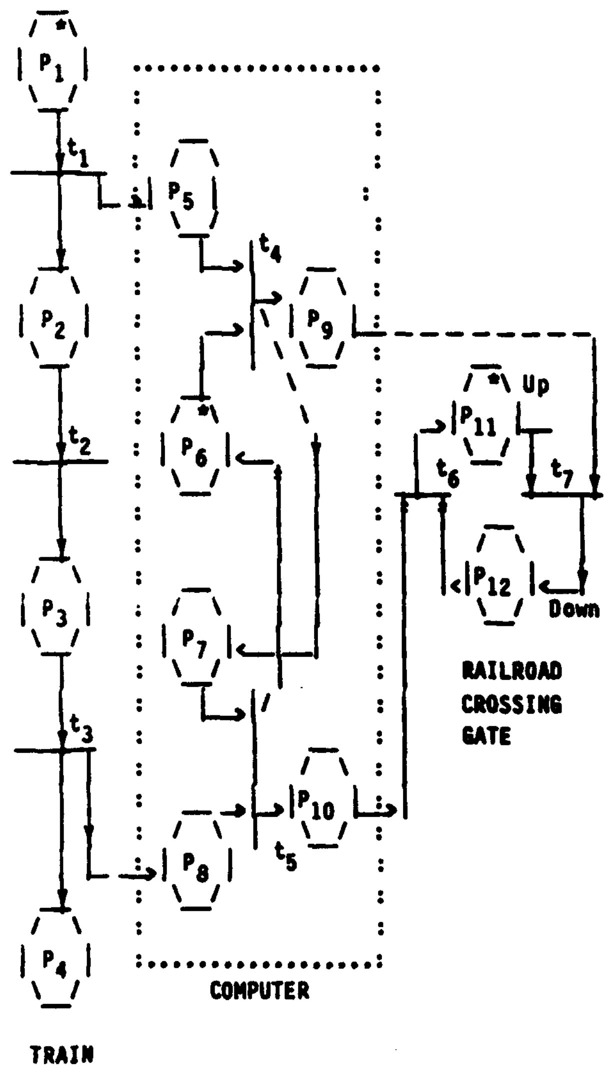


Figure 7.4.3

During the execution of the Petri net, the "state" of the system at any time consists of the set of conditions which contain tokens. Starting from some initial state, the Petri net model can be executed showing all the legal states (sets of conditions) that the system can "reach." This information can be depicted graphically in a "reachability graph," i.e. a graph that shows the possible state sequences from a given initial state. If the design is non-deterministic (i.e. the order of the transitions or events is not constrained by the design), then the graph will have branches for the different possible ordering of events. Figures 7.4.4 and 7.4.5 show a Petri net and the corresponding reachability graph for a simple model of a railroad crossing where the left part of the model is controlling the crossing guard gate (which is on the right). The reachability graph shows that the design is flawed (from a safety standpoint) since it is possible for the train to be and the crossing guard gate to be up at the same time. Note that it also shows the sequence of events which can lead to this hazard so that appropriate measures can be taken to avoid it.



P_1 = Approach P_2 = Before Crossing
P_3 = Within P_4 = Past

Figure 7.4.4. A Petri Net Graph

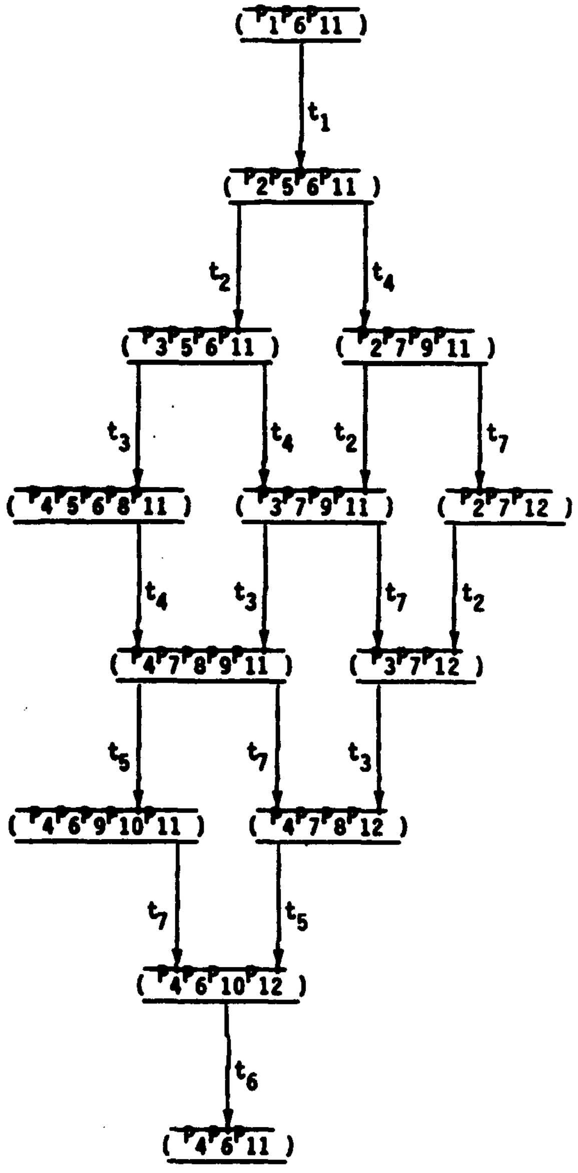


Figure 7.4.5. Reachability Graph for Figure 7.4.4

. .though creating the entire reachability graph will show whether the system as designed can reach any hazardous states, in practice the graph is often too large to generate completely. However, it is possible to use the same type of backward analysis used in fault trees. Algorithms have been developed at the University of California Irvine to do this backward analysis. Hazards which have been determined by the analysis to be plausible can be eliminated by appropriately altering the design to ensure that paths (sequences of events) which will lead to the hazard are not taken. For example, an interlock to ensure that one transition or event always preceeds another is shown in Figure 7.4.6. Figure 7.4.7 shows the Petri net equivalent of a lockout (i.e. a design feature that ensures that two conditions, in this case conditions P_3 and P_4, do not hold at the same time).

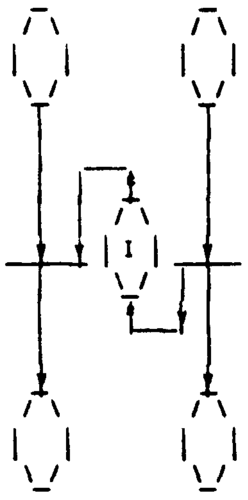


Figure 7.4.6.
Interlock

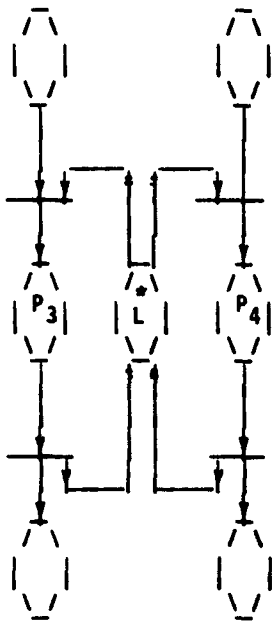


Figure 7.4.7.
Locking Place

Faults and failures can be incorporated into the model to determine their effect on the system. Backward procedures can determine which failures and faults are potentially the most costly and therefore which parts of the system need to be augmented with fault-tolerance and fail-safe mechanisms. Early in the design of the system, it is possible to treat the software parts of the design at a very high level of abstraction, i.e. only failures at the interfaces at the interfaces of the software and non-software components may be considered. By working backward to this software interface, it is possible to determine the software safety requirements and to determine which functions are most critical. Timing has been added to Petri net models by putting minimum and maximum time limits or transitions or by putting times on conditions. Either way, it is possible to determine worst case timing requirements so that, for example, watchdog timers can be incorporated into the design if necessary. Finally, when probabilities are included in the model, minimal cut sets and other probabilistic information is obtainable.

Although Petri nets have been used since the 1960's, the application of the technique for safety is still experimental (as of 1985) and has had little real-world validation. However, it appears to be a promising tool worth further consideration.

8. <u>SOFTWARE SYSTEM SAFETY DESIGN RULES AND GUIDELINES</u>

8.1 This section is designed to act as a guide in providing safety requirements for system and software specifications. It is not a definitive list nor can it be applied blindly. The list should be carefully reviewed and tailored to reflect the specific applications of the system(s) under development.

8.2 The intent of the the following rules and guidelines is to carry out the cardinal safety rule and its corollary that no single event or action shall be allowed to initiate a potentially hazardous event and that the system, upon detection of an unsafe condition or command, shall inhibit the potentially hazardous event sequence and originate procedures to bring the system to a predetermined "safe" state.

8.2.1 Documentation of Requirements, Designs, and Intents

8.2.1.1 All safing requirements shall be detailed in the software design specification.

8.2.1.2 The system shall make use of self test, BIT, and fault tolerant concepts.

8.2.1.3 Interrupt priorities and responses shall be specifically documented and analyzed for safety impact.

8.2.1.4 Techniques for deadlock prevention/detection/resolution shall be provided.

8.2.1.5 The intent of the design must be clearly stated and how it is to be accomplished in the target code. This includes, but is not necessarily limited to, absolute/relative time sequence, bit patterns, multiple data checks, and a priori tasks performed.

8.2.1.6 Critical software functions must not be corrected by patching in a language other/lower than the source code designated for that software. All patches must be clearly documented for meaning and logged with configuration control to maintain accurate change trail records for customer acceptance and delivery.

8.2.1.7 Any and all assumptions concerning unknown or unstated requirements and/or interfaces at any level of hardware or software must be clearly stated for the target code. This includes "TBD" requirements/interfaces as well as missing links/unspecified items.

8.2.1.8 Any and all deviations and waivers to the requirements and/or interfaces at any level of hardware or software must be clearly stated for the target code including implications or explanations to limitations.

8.2.1.9 The operating system (OS) software, as much of it that is used in the total software, must be clearly described, including shareable re-entrant library-type of routines and primitives. The portion of this OS software used by the critical software must be clearly identified as to the purpose for which it is used.

8.2.2 Critical Software/CPU/System Design Rules

8.2.2.1 Any single CPU controlling a process which could result in major system loss, damage, or loss of human life shall be incapable of satisfying all the requirements for initiation of a critical process. In a multiple CPU environment, two or more CPU's shall be required to initiate such a process. In a single CPU environment, other/dedicated hardware logic devices shall be AND'ed with the CPU commands to initiate such a process.

8.2.2.2 Critical data communicated between CPU's shall successfully pass data transmission verification checks in both CPU's.

8.2.2.3 The results of critical algorithms shall be verified prior to developmental and tactical use.

8.2.2.4 Critical software design and code shall be structured to enhance comprehension of decision logic.

8.2.2.5 Software design and code shall be modular in an effort to reduce logic errors and improve logic error detection and correction functions.

8.2.2.6 Memory locations shall have unique references (e.g., one and only one name) to prevent memory useage conflicts between global/local variables.

8.2.2.7 The system executive software shall maintain system control at all times.

8.2.2.8 Hazardous processes requiring critical timing shall be automated. (Refers to event sequences which must be tightly controlled and/or beyond human response time.)

8.2.2.9 Operations employing time limits impacting system safety shall have these time limits included in the software decision logic. Both relative time and mission/absolute time limits must be included when specified and used as part of the design.

8.2.2.10 The safety-critical time limits in the software logic shall not be changeable by an operator/flight officer from a console/cockpit.

8.2.2.11 Upon detection of an anomaly, the system/weapon shall revert to a safe configuration.

8.2.2.12 Upon detection of a critical anomaly, the software shall inform the operator what anomaly was detected. Non-critical anomalies shall be recorded or reported by telemetry to aid in system analysis.

8.2.2.13 Upon detection of a critical anomaly, the software shall inform the operator what action was taken. System response to non-critical anomalies shall be available for system analysis.

8.2.2.14 Upon safing the system, the software shall identify the resulting system configuration or status to the operator.

8.2.2.15 Workaround procedures shall not be allowed when reverting to a safe configuration after the detection of an anomaly.

8.2.2.16 Safing scenarios for safety critical hardware items shall be designed into the logic.

8.2.2.17 The software system shall protect against unauthorized access.

8.2.2.18 Software-controlled sequences affecting safety shall require a minimum of two independent procedures for initiation.

8.2.2.19 There shall be procedures to safely handle contingencies.

8.2.2.20 Interrupt priorities and responses shall be specifically defined and implemented in the software.

8.2.2.21 The software shall be initialized to a known and predictably safe state, from the program counter initial value through program execution to initialize data memory values.

8.2.2.22 The software shall terminate to a known and predictably safe state. All settable, non-volitile devices and relays shall be set to a known, safe state in anticipation of system restart or random power inputs/outputs.

8.2.2.23 All unused memory locations shall be initialized to a pattern that if executed as an instruction will cause the system to revert to a known safe state. (For example,(assuming a Z-80 processor) any entry into a pattern of NOP,NOP,JMP,NOP,NOP,JMP,NOP,NOP.... would cause a jump to address 0000, where a system safety/ restart routine could reside.)

8.2.2.24 There shall be provisions to protect the accessibility of memory region instructions and/or data dedicated to critical software functions.

8.2.2.25 There shall be provisions to protect instruction types in any program memory region from being used by any except critical software functions. (Instructions, for example, which can only be used in a supervisor mode.)

8.2.2.26 No software shall use a STOP or HALT instruction or cause a CPU WAIT state. The CPU must always be executing, whether idle with nothing to do or actively processing.

8.2.2.27 Flags shall be unique and shall be single purpose.

8.2.2.28 Files shall be unique and shall be single purpose.

8.2.2.29 CPU's that have simple instruction sets are preferred to CPU's that offer extensive instruction variants.

8.2.2.30 CPU's that load entire instructions and data words are preferred to CPU's that multiplex instruction and data words in n-parts.

8.2.2.31 CPU's that segregate instructions and data memories in separate banks served by separate (and different width) busses are preferred over CPU's that have a single memory bus to access both instructions and data from the same address space.

8.2.2.32 A CPU that time-shares its memory bus(es) between address and data functions should provide at least TBD complete clock cycles between functions on each bus.

8.2.2.33 Critical operational software instructions must be resident in non-volatile read-only memory (ROM). If a safety kernal is used, it also must be resident in ROM.

8.2.2.34 Data used by critical software should be protected by error checking and correcting (ECC) memory/memory controller.

8.2.2.35 Fixed or one-time changing data used to vector critical software should reside in EAROM, EEROM, or similar latching memory for retention permanence. The write control of such memory shall be considered critical and shall be treated in the same vein as a potentially hazardous event.

8.2.3 CPU-Hardware/Hardware-CPU Interfaces

8.2.3.1 A status check of critical one-shot system elements shall be made prior to executing a potentially hazardous sequence. (Safe-arm-device (SAD) status, interlock status, etc.)

8.2.3.2 An operational check of testable critical system elements shall be made prior to executing a potentially hazardous sequence. (Time clocks, door operation, batteries up, CPU execution, umbilical disconnected, etc.)

8.2.3.3 Upon completion of a test where safety interlocks were removed, the restoration of those interlocks shall be verified.

8.2.3.4 When software generates a hardware command, a design analysis shall be performed to determine if the command should be continuous until reversed or only used/applied for a discrete length of time. Reason: Potential hazardous timing, sneak timing, etc.

8.2.3.5 Decision logic using registers which obtain data values from end item hardware/software shall not be based on values of all "ones."

8.2.3.6 Decision logic using registers which obtain data values from end item hardware/software shall not be based on values of all "zeros."

8.2.3.7 Decision logic using registers which obtain data values from end item hardware/software shall require a specific binary data pattern of 'ones' and 'zeroes' to reduce the likelihood of malfunctioning end-item hardware/software satisfying the decision logic.

8.2.3.8 Separate "arm" and "fire" commands shall be required for ordnance initiation.

8.2.3.9 Critical hardware controlled by software shall be initialized to a known safe state.

8.2.3.10 Unless specifically exempted, input/output ports shall not be used for both critical and noncritical functions. Relates primarily to CPU's with two or more dedicated input/dedicated, output/dedicated, input-output/combination ports and special I/O instructions to access them.

8.2.3.11 Critical input/output ports shall have addresses sufficiently different from non-critical ports that a single (multiple) address bit failure will not allow access to critical functions or ports. Relates primarily to CPU's with memory-mapped I/O but can also apply to CPU's with a single combination I/O port and instructions to access it.

8.2.4 Other Hardware Factors/Utilities/Outside Influences

8.2.4.1 The system shall provide for a fail-safe recovery from inadvertent instruction jumps.

8.2.4.2 The computer system shall be immune to the effects of temporary power outages.

8.2.4.3 The computer system shall be sufficiently protected against the harmful effects of electromagnetic interferences.

8.2.4.4 The computer system shall be sufficiently protected against the harmful effects of electrostatic interference.

8.2.4.5 There shall be periodic memory integrity checks (of both program and data memories if separate or served by different busses).

8.2.4.6 The integrity of the instruction memory ROM should be enhanced with an ECC memory controller or a periodic software/hardware checksumming algorithm.

8.2.4.7 There should be periodic data and program memory bus operational checks.

8.2.5 (Interrupt) Operating System/Utilities Software

8.2.5.1 The software shall discriminate between false and valid interrupts.

.2.5.2 Operating System (OS) software, working on a fixed stack size with accumulated depth in applications use, shall not get lost in the sense of overflowing and overwriting other program instructions or data or in the sense of overflowing and not being able to correctly return all previous calls.

8.2.5.3 Operating System software shall place all shareable CPU resources on the stack/other prescribed memory area before a called module begins to execute with these resources. In most applications, this requires a full and complete verification of the compiler and/or assembler process tool to produce such machine code for the source code writer. In some cases, this OS software is coded once and replicated/called from each interrupt executive to handle any and all interrupt level software requests.

8.2.5.4 Re-entrant and shareable software utilities, such as trigonometric functions, primitives, etc, shall not overwrite their temporary scratch memory areas when they are used in several levels of prioritized interrupt processes.

8.2.5.5 Re-entrant and shareable software utilities shall not internally call another utility. For example, the SIN function shall not call a SQRT function and a TAN function shall not call a LOG function.

8.2.6 Operator Interfaces

The following items generally do not apply to embedded weapon software, such as software embedded in an airborne guided missile or a smart bomb. The operator interface is external to the weapon itself. The operator does have control over the weapon until the 'pickle' button is depressed, but from that time on, safety of the weapon operation rests entirely on the internal and autonomous weapon design. Notable exceptions would be weapons undergoing tests where a command destruct function is provided.

8.2.6.1 The software shall require two or more operator responses for initiation of any potentially hazardous sequence. No hazardous sequence shall be initiated without operator intervention and that sequence shall not be initiated by a single operator keyboard entry.

8.2.6.2 Operator interactions with the software shall be concise.

8.2.6.3 The software shall provide for detection of improper sequence requests by the operator.

8.2.6.4 The software shall provide for notification of improper keyboard entries by the operator.

8.2.6.5 The software shall provide for operator cancellation of current processing.

8.2.6.6 Operator cancellation of current processing shall require a single keystroke operator response.

8.2.6.7 Cancellation of processing shall be accomplished in a safe manner.

8.2.6.8 Any override of a safety interlock shall be identified to the test conductor via display on the test conductor's CRT.

APPENDICES

NOTE: The following appendices are **EXAMPLES** and are **NOT** designed to be used as written, **NOR** do they imply any preferred analysis methodology! They must all be carefully tailored to reflect the analysis method(s) desired and the system on which it(they) are to be applied!

APPENDIX A. SOFTWARE SAFETY ANALYSIS SPECIFICATION - EXAMPLE

NOTE: The following is strictly an **EXAMPLE!** It is **NOT** designed to be used as written **NOR** is it implied that sneak analysis is the preferred analysis methodology! An actual software safety analysis specification must be carefully tailored based both on the technique(s) involved and the system to which it is to be applied.

Purpose. The sneak analysis technique described herein establishes possible procedure for performing the analysis and reporting the results. The analysis is used to systematically identify and report sneak conditions. Document errors discovered during the sneak analysis are also reported.

Usage. Sneak analysis is a powerful tool to identify system conditions that could degrade or adversely impact mission safety or basic equipment reliability. It is a topological, graphical technique that can be applied to both hardware and software. The purpose of software sneak analysis is to define logic control paths which cause unwanted operations to occur or which bypass desired operations without regard to failures of the hardware system to respond as programed. After a sneak circuit analysis and a software sneak analysis have been performed on a system, the interactions of the hardware with the system software can readily be determined. The effect of a control operation that is initiated by some hardware element can be traced through the hardware until it enters the system software. The logic flow can then be traced through the software to determine its ultimate impact on the system. Similarly, the logic sequence of a software-initiated action can be followed through the software and electrical circuits until its eventual total system impact can be assessed. Finally, the analysis should be considered for critical systems and functions where sneak conditions cannot be tolerated, sneak conditions are suspected, or an unusually large number of anomalies have been observed to occur without any reasonable justification.

Applicable Documents: (Software Sneak documentation is pending and the following documents apply to hardware SCA only. They would need tailoring for use on software systems.) The following documents of the issue noted or, if not noted, the issue in effect as of the date of the contract as shown in Department of Defense Index of Specifications and Standards form a part of this specification.

TABLE 1. APPLICABLE DOCUMENTATION

DOCUMENT NUMBER	DOCUMENT NAME
1. MIL-STD-785B (15 September 1980)	Reliability Program Plan for Systems and Equipment
2. MIL-STD-781C (21 October 1977)	Reliability Design Qualification and Acceptance Test Standard (App. A, para 40.7)
3. MIL-STD-882B (30 March 1984)	System Safety Program Requirements
4. NAVSAEA TE001-AA-GYD-010/SCA	Contracting and Management Guide for Sneak Circuit Analysis
5. DOD 5000.40 (8 July 1980)	Reliability and Maintainability (para D.2a)

Requirements: During the full-scale engineering development and/or unlimited production phase, the contractor shall conduct or contract for sneak analysis of circuitry/software critical to mission success or crew safety. The analyses shall identify latent paths which cause unwanted functions to occur or which inhibit desired functions. Potential design or equipment weaknesses are to be identified and reported. An assessment of the system impact is to be provided for the potential problem along with a recommendation for corrective action. In making these analyses, all components/software shall be assumed to be functioning properly. These analyses shall be performed on the actual code and specifications for the software to be analyzed during the full-scale engineering development of later phases. Equivalent or design type drawings or logic flows shall be used during earlier development phases.

The contractor shall present in the proposal a complete list or description of the functions/circuitry/software for which sneak analysis is to be conducted. The list of those functions/circuits/software to be analyzed shall be presented to the procuring activity together with the rationale for any deviations to the specified systems.

The contractor shall specify the overall task period of performance along with subtask periods of performance. Periodic reviews or report periods shall be established to promote timely transmission and consideration of contractor reports. A final report documenting the task and all findings shall be prepared, including the network tree database. A final report for the baseline analysis shall be required, followed at the conclusion of the change analysis task with a final report documenting the change analysis reports.

The contractor shall indicate the level of the sneak analysis task in the proposal. The typical software task occurs at the instruction level where cause and effect relationships are studied in detail. Systems defined as within scope of the contracted effort should be analyzed to the detail component level. An interface analysis should be performed for that portion of out-of-scope software directly interfacing with the in-scope systems.

Technique. Four classical categories of the sneak analysis technique shall be addressed:

a. Sneak paths, which allow software logic to flow along unsuspected routes.

b. Sneak timing, which causes functions to be inhibited or to occur unexpectedly as a result of timing or function sequencing.

c. Sneak labels, which cause an operator to initiate incorrect stimuli.

d. Sneak indicators, which produce ambiguous or false displays.

A formal sneak analysis shall involve: (1) classifying basic circuit or software recognition patterns into which the system elements fall, (2) application of "clues" or sneak condition criteria, applied to these patterns to uncover sneak conditions, (3) assessing the effect of the sneak conditions on system performance, (4) establishing accept/reject criteria, and (5) reporting of results to the procuring activity.

The contractor shall identify latent flow paths, unexpected operational modes, unnecessary components, etc., in case of hardware; or unused and inaccessible paths, improper branch sequencing, undesirable loops, etc., in the case of software.

The contractor shall document the criteria, assumptions, delineation of sneak paths, etc., of the analysis. The data shall be maintained and updated to reflect any changes in equipment configuration, if funded by contract.

Assumptions. Assumptions are made when performing sneak analysis to establish the analysis boundaries, define terminology, and keep the scope within cost-effective bounds.

TABLE 2. SOFTWARE SNEAK ANALYSIS ASSUMPTIONS

A. THE SOFTWARE SPECIFICATION IS THE DESIGN INTENT OF THE SOFTWARE.

B. THE ASSEMBLER OR COMPILER DOES NOT INTRODUCE ERRORS INTO THE SOFTWARE.

C. ASSEMBLED OR COMPILED SOFTWARE IS FREE OF SYNTAX ERRORS, I.E., TYPOGRAPHICAL ERRORS.

D. THE DATA PROVIDED REPRESENT THE COMPLETE SOFTWARE PROGRAM UNDER CONSIDERATION.

E. HARWARE-INDUCED PROBLEMS ARE NOT CONSIDERED.

Quality Assurance Provisions. To insure that a valid sneak analysis will be accomplished, provisions must be established to provide that: (1) all paths within a system have been analyzed, (2) each path is directly traceable to the network tree in which it was analyzed, (3) each component/statement is directly traceable to the path in which it was analyzed, and (4) each component/statement is directly traceable to the specific documentation used to establish the data base master file.

The following provisions shall be used to produce a valid sneak analysis:

1. Network trees analyzed for sneak conditions shall be traceable to the system's source code. The network trees shall contain all statements used to generate the tree. Further, all paths necessary to initiate and complete a given function shall be shown or referenced on one network tree.

2. Cross-references listing data content and allowing traceability of the network trees shall be furnished.

3. All software network trees, completely annotated to show code represented, shall be furnished as part of the output of this analysis. These trees should be in a neat and understandable format and shall represent a self-contained data base from which other analyses (software fault tree, software hazard) may be done.

4. Each path shall be independently analyzed as to its effects on system operation and records maintaines indicating analysis results.

APPENDIX B. REQUEST FOR PROPOSAL EXAMPLE

NOTE: The following is strictly an EXAMPLE! It is NOT designed to be used as written NOR is it implied that sneak analysis is the preferred analysis methodology! An actual software safety analysis request for proposal must be carefully tailored based both on the technique(s) involved and the system to which it is to be applied.

Sneak Analysis Request for Proposal Considerations. The basic requirements for a procuring activity-initiated sneak analysis request for proposal (RFP) are presented in this section. The outline can be tailored by the procuring activity to suit a specific application. The outline is intended to be specific enough for the procuring activity to properly assess and evaluate contractor responses. Evaluation criteria are also presented to provide a basis for evaluation. The outline of the sneak analysis RFP and evaluation criteria are shown in Table 1.

TABLE 1. OUTLINE OF A SNEAK ANALYSIS RFP PROPOSAL

REQUEST FOR PROPOSAL (RFP) OUTLINE

1. PROGRAM NAME	8. DATA REQUIREMENTS
2. PURPOSE OF RFP	9. TASK DESCRIPTIONS
3. SCOPE OF EFFORT	10. DELIVERABLES
4. APPLICABLE SUBSYSTEMS	11. MISCELLANEOUS
5. ANALYSIS DEPTH	12. FACILITIES AND SECURITY
6. CHANGE ANALYSIS OPTION	13. COST
7. ADDITIONAL ANALYSIS	14. TIME REQUIREMENTS

EVALUATION CRITERIA (EC)

1. UNDERSTANDING PROBLEM	4. COST
2. RELEVANT PAST PERFORMANCE	5. SCHEDULE
3. CAPABILITY TO PERFORM	

Request for Proposal Items:

1. Program Name - This is a title of the sneak analysis task which is generally the program name. The major subsystem equipment or software name may be added to the title.

2. Purpose - The purpose(s) for requesting the sneak analysis should be stated. The rationale may be oriented toward problem identification of design analysis at the validation and full-scale engineering development phases or problem identification or change analysis in later development phases. This section should stipulate the task as being a "one-shot" analysis, a continuing analysis with change analysis included, or a combination of sneak analysis with one or more analysis techniques. Combined hardware and software sneak analyses techniques are to be stipulated as an integrated analysis (system, interface, detailed).

3. Scope of Effort - The task scope should delineate task requirements, depth of analysis, system or subsystems included in the analysis, period of performance, and deliverable end products. The scoping paragraphs may contain important notes or clauses from the remaining RFP items described in this section. Specific systems, subsystems, or portions of the software may be excluded from the effort and listed in this section. If multiple systems or subsystems are to be analyzed one at a time, the order and time phasing for each subtask should be specified. The responsibility for data acquisition should be identified as a task for the procuring activity, the sneak analysis contractor, or a third party. When classified systems are to be analyzed, security requirements should be specified for personnel, facilities, documentation handling procedures, and computer processing. The task scope should specify whether a change analysis is included in the overall effort, and if so, the number and type of acceptable changes should be specified.

4. Applicable Subsystems - Applicable software is to be accurately and completely identified in the RFP. Figures and pictures may be used to clarify and bound the applicable areas. Accuracy is required in this item because cost, schedule, or problem identification limitations may require analysis of only portions of particular subsystems and whole subsystems in others. Whenever portions of primary functions in the applicable subsystems continue into equipment or software not considered in scope, it is necessary to provide interface control documents, functional system diagrams, or logic-type diagrams that identify or depict the remainder of the function.

5. Analysis Depth - Analysis depth is an important scoping consideration because it has a direct bearing on cost, schedule, and antic-ip ced results. The procuring activity should specify the level of the sneak analysis required as system, interface or detail level. This RFP item is important to the procuring activity because it enables a correlation RFP response. Some responses may cover the designated software for markedly lower cost and schedule time due to performance of a higher level systems analysis rather than a more detailed systems analysis. The analysis depth specified in the RFP must be matched to the level of detail in the acquired documentation.

6. Change Analysis Option - The incorporation and analysis of software code instruction changes must be specified in the RFP if desired by the procuring activity. The change option may be limited to an analysis for only the proposed software design changes brought about by prime contractor response to sneak analysis reports. A more formal change analysis would include all changes to a particular configuration baseline. In either case, the procuring activity should specify the number and type of changes desired and the change analysis period of performance.

7. Additional Analyses - Whenever sneak analysis is to be performed in conjunction with other analyses, the selection and phasing of the tasks should be specified. Care should be exercised in this process to insure that the maximum benefits are achieved for each of the analyses. Combined analyses are usually performed on highly criticality software. They may also be applied to areas which have been manifesting anomalies. Combining analyses is also a means of achieving cost reductions. Central to any analysis is the effort required to establish the configuration on which the analysis is to be performed. A combined analysis effort requires the design configuration building only once, and this has typically been accomplished by the sneak analysis task for electrical and software systems. The sneak analysis network tree approach provides a functional layout of the system where cause and effect relationships can be identified and depicted easily. While the fundamental premise of sneak analysis assumes no equipment or software failures, failures can be postulated at any level and their effects traced through the systems in the same way as a detailed failure mode and effects analysis. The network tree data base also serve as the basis for other analyses, which eliminates duplicate efforts, standardizes the configuration, and lowers the overall costs.

8. Data Requirements - Analysis requires compilable source code listings, system block diagrams. high-level functional diagrams and program description documents, including requirements and specifications. Also, ;any logic flow diagrams, flowcharts, machine specific procedures, system architecture, and program environment information is needed where applicable.

The procuring activity must assign the data acquisition task responsibility to the prime contractor and subcontractors to the procuring activity itself, or to the performing sneak analysis contractor. The procuring activity, working through the prime contractor and vendors, has been the customary designee for this task. Allocations for the data acquisition effort will be made regardless of who is selected. Items to be considered for this effort are the prime contractor and vendor costs and delivery schedules if they are designated for data acquisition. The procuring activity and the sneak analysis contractor require funding to identify, order, and possibly travel to acquire the documentation.

The procuring activity should verify that the overall program contract has requirements for prime contractor, subcontractor and vendor deliveries of software code and documentation which will permit sneak analysis to be performed in a timely and cost effective manner. If delivery of code documentation and code has not been contracted for in the overall program procurement process, additional cost will be incurred by the procuring activity in obtaining data for the analysis. Third party agreements will also have to be established which identify the documentation users, the purpose, data handling procedures, period of usage, and final disposition of the documentation.

Documentation acquisition must be timely and complete early in the sneak analysis task or schedule impacts will occur. The majority of sneak analysis tasks are under 9 months in duration which means that the complete documentation for such a task should be received within a month of task initiation. Definite milestone dates indicating 50 percent and 100 percent levels for data acquisition should be stated in the RFP.

The RFP should specify when the analysis is to include classified systems so that special handling procedures are instituted. Personnel availability with proper security clearance levels must be established, along with facility clearances. If data processing is to be performed using classified data, then the computer and computer facility must have been approved for classified data processing.

9. Task Descriptions - The RFP should stipulate that the competing sneak analysis contractors provide a description of the tasks they are to perform to identify sneak conditions in software. The approach should be systematic, thorough, and complete. The task description should also declare whether any automation aids are contemplated for use in accomplishing the sneak analysis task. Descriptions of the automation aids should be provided including the rationale for use. Relevant points to consider for automation include:

a. Direct conversion of prime contractor manufacturing data from magnetic tape (or other suitable media) to acceptable input format for the sneak analysis computer programs.

b. Generation of the network trees used in the analysis phase.

c. Change analysis.

10. Deliverables - The deliverables of a sneak analysis task should be specified in the contractor proposal, along with concise descriptions of each item. As a minimum, the specified reports and the network tree data base should be furnished along with any data cross references generated. As a minimum, the following output reports should be included:

a. Periodic Status Report - This report documents the progress of the various sneak analysis subtasks, identifies any problems encountered in the analysis which might impede successful completion of the project, tabulates all sneak reports issued, and includes copies of the sneak reports as they are issued. The status report is the primary means of conveying the findings of the task in a timely fashion. Requirements for either bi-weekly or monthly status reports should be specified. For long-duration projects special quarterly reports which summarized the activities of the previous 3 month period may be required.

b. Sneak Analysis Reports - This category of reports includes software sneak reports, software design concern reports, and software drawing error reports. The primary intent of each report is to document problems found in a form that is understandable, clear, concise, and easily verifiable. The paperwork should serve as a link from the sneak analysis contractor to the procuring activity and on to the prime contractor. The sneak analysis report should briefly describe the undesirable condition and reference relevent documentation.

c. Final Report - This report summarizes the entire activities of the sneak analysis task. The report should include the project purpose, scope, and an overall evaluation of the analyzed system(s). A complete listing of all documentation used in the analysis should be provided so that the configurations for the baseline system and system changes can be established. A brief description of the project tasks should be included. A tabulation of all reports issued by the sneak analysis contractor to the procuring activity should also be included, along with copies of each report. Any problems which had an impact on the successful completion of the task according to the scope and terms of the originating RFP should also be documented in the final report.

d. Network Tree Data Base. All hierarchical network trees generated during the analysis, drawn in a neat and well annotated manner, showing the code represented by the tree and completely relatable to the software being analyzed.

The three report types just presented are the minimum deliverables for a sneak analysis task. If change analysis is to be performed, any problem conditions can be reported sufficiently with the above type reports. If a verification of checklists, technical orders (TOs), or other military-type procedure lists are included within the scope of the contracted effort, slightly modified versions of the sneak analysis reports may be desirable. In this instance, the reports would identify operator-induced errors, errors in control sequencing, and errors of interpretation and reaction to equipment displays. Additional reports generated to document the results of other analysis techniques should also be described and included in the final report. The final report could also be used to implement the recommendation to measure sneak analysis effectiveness.

11. Miscellaneous - Other uses of the data base should be reported separately. The requirements for the sneak analysis contractor over and above the baseline analysis should be specified. This may include, but not be limited to, the following:

a. Additional analyses, such as failure mode and effects analysis, fault tree analysis, and change analysis.

b. Data acquisition and travel.

c. Program review support to provide liaison and inputs for milestone events such as PDR, CDR, first flight, or scheduled tests.

d. Computer tool development, possibly in the area of converting prime contractor data to a format usable in the analysis phase.

e. Classified data handling.

12. Facilities and Security - Facilities and security requirements should be a part of the RFP, even when classified data systems are not involved. This feature protects the proprietary rights, if any, of the software owner. The entire contractor facility or some designated portion thereof should be cleared to handle the highest level documentation contemplated for use by the sneak analysis contractor. Personnel should have at least corresponding security clearances. This includes direct management, engineers and software analysts, clerks, secretaries, aides, and any other personnel participating in the classified portion of the analysis. If any portion of the task is computer-aided, then the personnel, computer, and computer facility must also be cleared. Disposition requirements to return or destroy acquired documentation and sneak analysis contractor-generated reports must be specified.

13. Cost - Cost breakouts for the various sneak analysis tasks will normally be a separate report or volume from the RFP to allow an impartial evaluation of the technical portion of the RFP. Cost factors include analysis personnel, computer processing, classified data handling, data acquisition, travel, performance of other analyses, and computer program development (if any).

14. Time Requirements - The time requirements (if any) of the procuring activity should be stated in the RFP. Dates for support of CDR, first flight, or major systems tests have a direct bearing on the tasks and sequences of tasks performed by the sneak analysis contractor. If the analysis is "one-shot" and scoped to a single system or portion of a single system, then the task approach is fairly direct. Otherwise, multiple systems analysis or combined analyses will require the contractor to prioritize the systems and the tasks within each system.

Evaluation criteria items:

1. Understanding of the task - Understanding of the task is an important evaluation criterion to judge sneak analysis proposals. The evaluation criterion looks not only for an understanding of the contractor's sneak analysis process and task relationships but also the need and use of the analysis to satisfy the procuring activity's requirements in a timely and cost-effective manner.

The task process may be different from contractor to contractor but will probably progress according to the following order:

a. Data Acquisition

(1) Identify and acquire data if so tasked.

(2) Log all documentation received.

(3) Review all documentation for completeness and correct level of detail.

(4) Identify missing and required documentation.

b. Preanalysis

(1) Identify functions in the documentation.

(2) Verify system interconnections.

(3) Exclude all areas of documentation out of scope for the effort.

(4) Review adequacy of interface equipment and/or software documentation.

c. Partitioning

(1) Subdivide or software by functions or structure.

(2) Annotate documentation for subsequent encoding task.

d. Encoding (only if computer processing is used)

(1) Convert detailed source code to computer format.

(2) Automate conversion process.

(3) Verify data master files reflect actual system configuration.

e. Computer Processing

(1) Edit and analyze all user inputs.

(2) Connect all and software code (and circuit segments if appropriate.)

(3) Produce hardcopy plots of system network trees.

f. Network Tree Generation

(1) Manually develop network trees if no automation aids are used.

(2) Annotate all functional remarks on network trees.

(3) Annotate all cross-references on network trees.

(4) Identify and annotate relevant descriptive documentation.

(5) Annotate interface information for out-of-scope systems.

g. Analysis

(1) Identify topographs (software patterns) in network trees.

(2) Apply "clues" for each topograph.

(3) Compare network trees to functional system flows.

(4) Compare network trees to interface control documents.

(5) Compare network trees to procedure checklists.

(6) Perform timing analyses where required (i.e., interrupt and data availability analysis).

h. Problem Reporting

(1) Assess problem categorization (documentation, design, sneak).

(2) Identify relevant documentation.

(3) Describe problem.

(4) Determine system impact.

(5) Provide sketch of system illustrating problem.

i. Status Reporting

(1) Provide periodic status reports (bi-weekly or monthly) and contract funds status reports (or equivalent) providing "dollar" information.

(2) Provide quarterly status reports, if required.

(3) Include all reports issued during the period.

(4) Tabulate and identify report dispositions.

j. Change Data Receipt (if change analysis option is included).

(1) Acquire proposed and/or implemented system changes.

(2) Log all documentation received.

(3) Group changes by function or configuration control board package numbers.

k. Change Data Incorporation

(1) Determine extent of change package.

(2) Update network trees.

l. Change Data Analysis

(1) Reanalyze all changed network trees.

(2) Reanalyze all changed network trees affected by the changed network trees.

(3) Rerun timing analyses, if necessary.

(4) Problem reporting is the same as baseline analysis.

m. Final Report

(1) Summarize task purpose and scope.

(2) Describe analysis technique.

(3) Provide composite listing of all documentation.

(4) Provide tabulation of all reports and dispositions.

(5) Provide copies of all reports.

(6) Provide copies of all reports.

(7) Provide network tree data base and data cross references.

2. Relevant Past Performance - Actual performance, along with an assessment of that performance, should be demonstrated and presented in the contractor's proposal. The distinction between actual performance and performance capability needs to be identified and evaluated in the contractor's response. To adequately evaluate responses, the contractor should be required to provide task synopses similar to those required by the application guidelines procurement and, in addition, provide, upon request, copies of final reports for procuring activity inspection. Resumes of key personnel should also be provided. Emphasis should be placed on showing experience in similar programs.

3. Capability to Perform - Capability to perform sneak analysis is the evaluation criterion which will be the most difficult to assess. Different approaches will be offered in competitive proposals by prospective contractors. Some may include the formal sneak analysis process, variations to the process, or substitution of other analysis techniques to achieve the same end results. As a minimum, the capability to perform the basic tasks listed in Item 1 would be a prerequisite for further proposal evaluation. Some of the main task headings may change, but the overall tasks should be equivalent. Current automation aids and descriptions of their intended use should certainly strengthen the evaluation rating in the capability to perform criteria.

4. Cost - Line item costs for the basic analysis, change analysis, and other analyses should be separately reported in the proposal. Costs for personnel, management, and clerical support should be delineated, as well as computer processing costs, travel costs, and any other miscellaneous cost items. In general, the greater number of executable instructions in the software, the greater the sneak analysis cost. However, cost needs to be evaluated against proposed depth of analysis and against the quality of the code. Well structured, modularized, and will documented code is cheaper to analyze.

5. Schedule - Contractor schedules should be evaluated for the proper sequencing and duration of tasks. They should identify all major task functions and project milestones. Since data acquisition is very critical, early milestones for data receipt are an absolute necessity. Network tree construction can either be shown as one complete task of relatively short duration in the first half of the project or as a continuing process throughout most of the period of performance. Either approach is acceptable. Analysis is the main task element

of the procurement. The longest duration task should be the analysis, but the schedule duration may not indicate such emphasis.

The contractor schedule should also include support of reviews, other analysis technique sequencing and duration, change analysis if selected, and final report preparation and delivery. If classified or proprietary data; are used, there should be a final data disposition task included.

Contract Selection: Selection of the type of contract desired for the sneak analysis procurement is primarily the option of the procuring activity and should be stipulated in the RFP.

APPENDIX C. STATEMENT OF WORK EXAMPLE

NOTE: The following is strictly an **EXAMPLE!** It is **NOT** designed to be used as written **NOR** is it implied that sneak analysis is the preferred analysis methodology! An actual software safety analysis statement of work must be carefully tailored based both on the technique(s) involved and the system to which it is to be applied.

Tailoring Statement of Work Requirements. The process for tailoring sneak analysis statement of work requirements to acquisitions of various types is shown in generalized form in Figure 1. The process assumes that an assessment of specification requirements and application guidelines resulted in a decision to perform the sneak analysis task. The need and the rationale for the task are thus established. The process flow is now directed toward determining how the procurement is to be implemented.

Tailor Process: The process flow shown will be presented as a step-by-step description of each function or decision block. The blocks are numbered to clarify the path direction within the decision tree.

1. Identify Analysis Areas - The process begins in block 1 with an identification of relevant systems or subsystems. On the initial pass, all software within each of the designated systems may be scoped to determine overall rough order of magnitude costs.

2. Estimate System Size - Block 2 requires an estimation of the number of instructions in the software system. Typical systems are composed of high-order languages, assembly languages, and machine code. Each instruction can be translated into an equivalent number of assembly language instructions.

3. Compute Cost - Based on the data from items 1 & 2 compute rough order of magnitude costs. Software quality and languages involved will greatly influence potential costs.

4. Acceptable Cost Range - The derived rough order of magnitude cost of the sneak analysis task is compared to the budget levels allocated in the original program reliability plan. If anticipated costs are within the budgeted level, the procuring activity may begin the procurement process by proceeding to the function in block 8 of Figure 1. However, since the original estimate was a rough order of magnitude, it may be necessary to perform a more detailed estimate of costs to achieve higher confidence in the cost figures.

If the task cost significantly exceeds the allocated analysis cost, additional decisions must be made by considering reduction of analysis scope or combining sneak analysis with another analysis.

5. System Scope Reduction - If the rough order of magnitude estimate exceeded the planned task allocation, a critical analysis of the designated systems should be performed to isolate the desired functions. In effect, the task effort should be scoped from the overall system level to the subsystem level and possibly even to the instruction level. Noncritical functions should be eliminated and a new scope of effort determined. The reduced task scope may then be evaluated against the original statement of need and costs recomputed by entry to block 2. Process cycling through scope reductions and cost computations may occur until desired task cost levels are achieved.

NOTE - The procuring activity could request contingency funds to supplement the original task allocations. Program contingency funds are those dollars not previously allocated to other analyses. Sufficient budget could then be obtained, and the process flow could proceed to block 8.

6. Combine Analyses - If it is not possible to reduce the number of functions scoped for the analysis task, the procuring activity should consider combining sneak analysis with other analysis tasks. The overall cost of performing sneak analysis with specific analyses, such as failure mode and effects analysis and fault tree analysis, is lower than performing all three tasks separately. In this way, lower overall budget expenditures can be obtained for the combined tasks. The resulting allocations should be sufficient for the combined tasks. If the resulting budget allocation is sufficient, proceed to block 8; otherwise the sneak analysis task must be deferred until supplemental funds are available or the procurement canceled. An option is available to the procuring

activity that removes the requirement for an interface sneak analysis. Very early in the program development phase (concept or validation phase) this may be a viable alternative. From the full-scale development phase and later phases, the analysis should be at the detailed level to obtain significant results.

7. Defer Procurement - If this task decision block is reached, a reconsideration of rationale or statement of need should be made. If the decision is made to defer the task at the earliest available time. <u>The cost to identify problems, correct design, and implement changes increases dramatically when problems are detected in later project development phases.</u>

8. Contract Selection - After the sneak analysis task is properly scoped and program funds are allocated, the next function to be performed is selection of the type of contract and statement of work. The selection of the proper statement of work is dependent on the software composition and the addition of other task analyses.

9. Competitive Procurement - The decision to issue a sole-source procurement or a competitive-bid procurement can probably be made at an earlier time in the process flow. If sole source selection is desired, proceed to block 13 of the process flow. If a competitive-bid procurement is selected, the decision should be made earlier in the development flow to overcome the attendant delay in the project initiation schedule until contract award.

10. Issue RFP - The bid package should then be circulated and announced in publications such as Commerce Business Daily.

11. Evaluate Responses - Evaluation criteria considerations were presented earlier and may be used as the basis for evaluating contractor responses. Care should be taken not to bias in favor of any one methodology, so long as a topological/graphical approach is maintained.

12. Contract Award - Contract award to the winning bidder is the final step in the competitive procurement process and represents the initiation of the sneak analysis task.

Figure 1. Tailoring Statement of Work Requirements

STATEMENT OF WORK FOR SOFTWARE

1. GENERAL

A software sneak analysis shall be performed on the ____________________ system. The analysis shall be performed using topological/graphical techniques as referenced in section 3.4.2.2. This analysis shall identify data paths and logic conditions that can cause an unwanted function to occur or inhibit a desired function without a hardware failure. Recommendations for corrective action to eliminate these conditions shall be provided. Also, document errors and areas of design concern discovered during the analysis shall be reported.

2. SCOPE

The software sneak analysis shall cover the software programs for use by the ________________ system computer(s). The software shall be analyzed at the executable code/instruction level. The software program is coded in _________.

3. CHANGE ANALYSIS (OPTIONAL)

The analysis shall include changes due to corrective action taken to eliminate sneak analysis identified problems received prior to ____________, 19_____. The change analysis shall be limited to a total equal to _____ per cent of the baseline design data base as established by actual count of software instructions added, deleted, or revised due to corrective changes. Wherever possible, all changes shall be submitted by copies of the completed software problem/change reports, copies of the revised data tapes or card decks, and listings.

4. TASK DESCRIPTIONS

Specific tasks to be performed as part of this analysis contract shall consist of the following:

4.1 Receive and set up files for the software program and any design and program documentation defining the operation and functions of the software to be analyzed.

4.2 Develop a software network tree data base.

•.3 Perform a sneak analysis on the resulting software network trees to identify potential sneak conditions, such as:

4.3.1 Sneak paths, which may allow data or logic to flow along an unexpected route.

4.3.2 Sneak timing, which may cause data or logic to flow or to inhibit a function at an unexpected time.

4.3.3 Sneak indications, which may cause an ambiguous or false display of system operating conditions.

4.3.4 Sneak labels, which may cause incorrect stimuli to be initiated.

5. REPORTS

Reports will be submitted in accordance with the contract CDRL.

5.1 Prepare sneak software reports (SSRs) on all sneak conditions found. Each report shall describe the sneak condition in detail. The SSR shall include a listing of the suspect software instructions where appropriate. Recommendations for appropriate corrective action shall be given along with reference to supporting documentation. The report forms include a section for the customer to indicate the action taken to resolve the condition being reported. All reports shall be appropriately dated, titled, and numbered for indexing and tracking.

5.2 Prepare software design concern reports (SDCRs) to describe certain items of concern with specific design implementation. The conditions to be reported include:

5.2.1 Questionable design practices.

5.2.2 Unnecessary software instruction.

5.2.3 Unused software instructions.

5.2.4 Specifications not met or not clear.

5.3 Prepare software document error reports (SDERs) on discrepancies found in the input data for the analysis. Each report shall identify the discrepant document and explain the error relative to referenced supporting documentation.

5.4 Prepare and submit activity reports to describe the work accomplished during the reporting period. The reports will include analysis progress, problems, recommendations, and results of meetings. The reports shall be submitted in accordance with the contract CDRL. The SSRs, SDCRs, and SDERs generated during the reporting period shall be attached. A tabulation of all previously submitted reports (SSR, SDCR, and SDER), including status, shall be attached to each activity report.

5.5 Perform analysis of software changes, provided a change analysis is elected, and submit appropriate reports.

5.6 Prepare and submit a final report containing a summary of the analysis effort, including the general analysis method used, the extent of the analysis, conclusions drawn, and recommendations based on the analysis. All SSRs, SDCRs, and SDERs written shall be included as well as the completely annotated network tree data base. One report shall be prepared at the completion of the baseline analysis, and if a change analysis is performed, the report shall be revised to include any additional reports resulting from the change analysis.

6. DATA REQUIREMENTS

The analysis will be based on the source listing and source code which should be provided in a machine readable format. If the analysis is to be based on a high-order language, then the source program listing, high-order sourc code, and an assembled program listing should be provided. All reference manuals for the computer, cross-assembler, language, and operating system should be made available. Also, time/cost may be saved if other program documentation be provided, such as module descriptions, flow diagrams and data structure definitions, so that the potential system impact for problems found can be more accurately assessed. The above data will be furnished ______________. All data must be delivered by _______ to enable a timely and accurate analysis. Delay in receipt of data will result in a day-for-day slide in the schedule, and the contract price shall be equitably adjusted to reflect additional costs, if any.

7. PERIOD OF PERFORMANCE

The period of performance for the sneak analysis of the ______________________________ system shall be months after receipt of input data necessary to establish the configuration as defined in paragraph 2, SCOPE. Change analysis shall be performed on all previously described changes received prior to ______________, 19__, and the final report shall be submitted by ___________, 19__, provided the contractor has acknowledged all reports by having signed each report and stated the action taken to correct the reported problem.